This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.
Design and Implementation of a Reliable Reconfigurable Real-Time Operating System (R3TOS)

Xabier Iturbe

A thesis submitted for the degree of Doctor of Philosophy

The University of Edinburgh

May, 2013
Most of the research reported in this thesis was funded by
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification. However, some ideas and figures included in this thesis have been previously published in [Iturbe et al., 2009], [Iturbe et al., 2010a], [Iturbe et al., 2010b], [Iturbe et al., 2010c], [Iturbe et al., 2010d], [Iturbe et al., 2011a], [Iturbe et al., 2011b], [Iturbe et al., 2011c], [Iturbe et al., 2011d], [Hong et al., 2011], [Iturbe et al., 2012], [Hong et al., 2012], [Ebrahim et al., 2012], [Torregg et al., 2012b], [Hong et al., 2013], [Ebrahim et al., 2013], [Iturbe et al., 2013a], and [Iturbe et al., 2013b].

[Signature]

Xabier Iturbe
Nire senideei, orain nirekin daudenei eta jadanik joan direnei

To my family, to those who are with me now and to those who have passed
Abstract

T
WENTY-first century Field-Programmable Gate Arrays (FPGAs) are no longer used for implementing simple "glue logic" functions. They have become complex arrays of reconfigurable logic resources and memories as well as highly optimised functional blocks, capable of implementing large systems on a single chip. Moreover, Dynamic Partial Reconfiguration (DPR) capability permits to adjust some logic resources on the chip at runtime, whilst the rest are still performing active computations. During the last few years, DPR has become a hot research topic with the objective of building more reliable, efficient and powerful electronic systems. For instance, DPR can be used to mitigate spontaneously occurring bit upsets provoked by radiation, or to jiggle around the FPGA resources which progressively get damaged as the silicon ages. Moreover, DPR is the enabling technology for a new computing paradigm which combines computation in time and space. In Reconfigurable Computing (RC), a battery of computation-specific circuits ("hardware tasks") are swapped in and out of the FPGA on demand to hold a continuous stream of input operands, computation and output results. Multitasking, adaptation and specialisation are key properties in RC, as multiple swappable tasks can run concurrently at different positions on chip, each with custom data-paths for efficient execution of specific computations. As a result, considerable computational throughput can be achieved even at low clock frequencies. However, DPR penetration in the commercial market is still testimonial, mainly due to the lack of suitable high-level design tools to exploit this technology. Indeed, currently, special skills are required to successfully develop a dynamically reconfigurable application.

In light of the above, this thesis aims at bridging the gap between high-level application and low-level DPR technology. Its main objective is to develop Operating System (OS)-like support for high-level software-centric application developers in order to exploit the benefits brought about by DPR technology, without having to deal with the complex low-level hardware details. The developed solution in this thesis is named as R3TOS, which stands for Reliable Reconfigurable Real-Time Operating System. R3TOS defines a flexible infrastructure for reliably executing reconfigurable hardware-based applications under real-time constraints. In R3TOS, the hardware tasks are scheduled in order to meet their computation deadlines and allocated to non-damaged resources, keeping the system fault-free at all times. In addition, R3TOS envisages a computing framework whereby both hardware and software tasks coexist in a seamless manner, allowing the user to access the advanced computation capabilities of modern reconfigurable hardware from a software "look and feel" environment. This thesis covers all of the design and implementation aspects of R3TOS. The thesis proposes a novel EDF-based scheduling algorithm, two novel task allocation heuristics (EAC and EVC) and a novel task allocation strategy (called Snake), addressing many RC-
related particularities as well as technological constraints imposed by current FPGA technology. Empirical results show that these approaches improve on the state of the art. Besides, the thesis describes a novel way to harness the internal reconfiguration mechanism of modern FPGAs to perform inter-task communications and synchronisation regardless of the physical location of tasks on-chip. This paves the way for implementing more sophisticated RC solutions which were only possible in theory in the past. The thesis illustrates R3TOS through a proof-of-concept prototype with two demonstrator applications: (1) dependability oriented control of the power chain of a railway traction vehicle, and (2) data-streaming oriented Software Defined Radio (SDR).

**Keywords**: fault-tolerance, dynamic partial reconfiguration, reconfigurable computing, on-demand computing, real-time, hardware virtualisation, reconfigurable operating system, multitasking, software defined radio.

**Word cloud:**
Acknowledgments

"Gizonen lana jakintza dugu: ezagutuz aldatzea, naturarekin bat izan eta harremanetan sartzea. Eta indarrak ongi errotuz, gure sustraiak lurrari lotuz, bertatikan irautea: ezaren gudaz baiteza sortuz, ukazioa legetzat hartuz beti aurrera joatea."

"Man’s duty is to knowledge: to learn and change, to be one with nature and achieve to reveal it. Creating the positive from the negation, and taking the contradiction by law, continue making progresses."

— Xabier Leite

After three and a half intense years, this research comes to an end. In this moment, I would like to dedicate some lines to all of the people who have played an important role to make this thesis possible. I really hope I do not forget anyone.

First of all, my most sincere greeting to Dr. Khaled Benkrid (formerly at University of Edinburgh and now at ARM Ltd.) and Professor Tughrul Arslan (University of Edinburgh), who trusted me and my research proposal and whose excellent supervision and knowledge was indispensable to turn my initial ideas into a thesis work. Besides, they have guided me in my first steps within the research community, introducing me to a number of prestigious researchers in the field.

I would like to especially acknowledge the collaboration received from two colleagues in the field of reconfigurable computing: Dr. Marco D. Santambrogio (Politecnico Milano and MIT) and Dr. Tobias Becker (Imperial College London).

I also acknowledge the revisions coming from the thesis examiners who helped to improve the quality of this document. It has been my pleasure and honour to spend some hours discussing about my research work with Professor Michael O’Boyle (University of Edinburgh) and Professor Steve Furber (University of Manchester). The codesign/compiler-oriented perspective given by Professor Michael O’Boyle was very interesting in light of making R3TOS better in the future, and I really appreciate the experienced comments given by such a computing “legend” as Professor Steve Furber.
Greeting words also go to Dr. Alberto Ruiz de Olano (IK4-Ikerlan) who gave me the opportunity to start this journey that was rounding up my mind since some time before. I must admit that the courage to enroll in this project came in part from Dr. Armando Astarloa (University of the Basque Country), who had transmitted me the enthusiasm for researching during my undergraduate student period. In addition, I acknowledge his assistance while I was conducting my thesis research.

I acknowledge all workers from IK4-Ikerlan who directly or indirectly have helped me. I especially thank Ana Isabel Martínez, Dr. Imanol Martínez, Antonio Pérez (now at Orona) and Dr. Jon Pérez for their support and confidence. Moreover, the latter three provided very interesting contributions with a view to their experience in the industry field. This paragraph would not be complete without a specific greeting to my colleagues in the Electronics department. Your uninterested contribution has been very valuable for my research. Eskerrik asko!

I am very grateful to Dr. Ahmet T. Erdogan, who helped me when I first arrived to the University of Edinburgh, guiding around the messy corridors in Alrick building as well as with the academic procedures. I would also like to thank my international colleagues in the System Level Integration Research Group (SLIG) of the University of Edinburgh. All of you have shown me that there are multiple ways to understand the life and, particularly, to face engineering problems. My especial thanks go for the people who have been more closely involved in my thesis research: Mr. Chuan Hong, Mr. Ali Ebrahim, and Mr. Raúl Torrego. Raúl is especially familiar to me as we are colleagues in IK4-Ikerlan as well.

I also thank David Ross for having shown me the real Scottish temperament, making me feel in Edinburgh as at home.

Last but not least, I heartily thank the love and comprehension received from my parents, Javi and Estibaliz, and friends. You have patiently endured my frustration in the difficult moments of my research, pushing me towards the successful completion of this thesis. Therefore, it is my wish that you all consider part of it as yours.
# Contents

Abstract \hspace{1cm} v
Acknowledgments \hspace{1cm} vi
Contents \hspace{1cm} xi
List of Figures \hspace{1cm} xvi
List of Tables \hspace{1cm} xvii
List of Algorithms \hspace{1cm} xviii
List of Acronyms and Abbreviations \hspace{1cm} xix

## 1 Introduction

1.1 Open Problem \hspace{1cm} 8
1.2 Objectives of the Thesis \hspace{1cm} 8
1.3 Achievements of the Thesis \hspace{1cm} 9
1.4 Outline of the Thesis \hspace{1cm} 10

## 2 Introduction to FPGAs and Dynamic Partial Reconfiguration

2.1 Digital Design and Implementation \hspace{1cm} 14
  2.1.1 FPGA-based Synchronous Design \hspace{1cm} 17
  2.1.2 Modular SoC Design \hspace{1cm} 20
2.2 Dynamically and Partially Reconfigurable Xilinx FPGAs \hspace{1cm} 23
  2.2.1 Evolution of Partially Reconfigurable Architectures \hspace{1cm} 24
2.3 Virtex-4 Architecture \hspace{1cm} 27
  2.3.1 Built-In Logic Resources and Hardware Primitives \hspace{1cm} 28
  2.3.2 Virtex-4 Architecture Mapping to Configuration Bitstream \hspace{1cm} 38
  2.3.3 Reliability Concerns \hspace{1cm} 51
2.4 Dynamic Partial Reconfiguration \hspace{1cm} 60
  2.4.1 Difference-based Reconfiguration \hspace{1cm} 61
5.3 Simulation Results ......................................................... 173
  5.3.1 Simulation Set-Up .................................................. 173
  5.3.2 Case A: No Damage in the Device ............................. 176
  5.3.3 Case B: Damage in the Device .................................. 178
  5.3.4 Case C: Snake Task Allocation Strategy ...................... 182
5.4 Chapter Conclusion .................................................... 182

6 Low-Level Hardware Support for the R3TOS HWuK 185
  6.1 R3TOS Static Infrastructure ......................................... 187
  6.2 Task Control Logic (TCL) ............................................. 190
    6.2.1 Functioning ....................................................... 190
    6.2.2 Implementation .................................................. 193
  6.3 Data Relocation Tasks (DRTs) ....................................... 199
  6.4 Scheduler ............................................................ 204
    6.4.1 Task BRAM ......................................................... 205
    6.4.2 The Scheduler’s PicoBlaze ...................................... 209
  6.5 Allocator ............................................................. 214
    6.5.1 FPGA State BRAM ................................................. 215
    6.5.2 Empty Area Descriptor Updater (EADU) ....................... 217
    6.5.3 Architecture Checker (AC) ..................................... 218
    6.5.4 Allocation Quality Evaluator (AQE) ............................ 221
    6.5.5 The Allocator’s PicoBlaze ...................................... 222
  6.6 Configuration Manager ............................................... 223
    6.6.1 Bitstream BRAM ................................................... 224
    6.6.2 ICAP Driver ....................................................... 231
    6.6.3 STARTUP Support Logic .......................................... 235
    6.6.4 Configuration Guardian (CG) ................................... 237
    6.6.5 CRC32 Module ..................................................... 238
    6.6.6 The Configuration Manager’s PicoBlaze ....................... 238
  6.7 HWuK Internal Communications and Functioning .................. 239
    6.7.1 Communication Monitor ......................................... 240
    6.7.2 R3TOS Implementation Key Points ............................. 242
  6.8 Chapter Conclusion .................................................. 243

7 Hardware Abstraction Layer: Low-Level R3TOS HWuK Services 244
  7.1 Task (Re)Allocation Service ......................................... 245
    7.1.1 (Re)Allocation of Partial Bitstreams in-between Clock Regions 247
  7.2 Task Deallocation Service ............................................ 251
  7.3 Task Preemption Service: Context Save and Restore ............. 252
  7.4 Inter-task Communication Service .................................. 255
  7.5 Clocking Management Service ....................................... 258
  7.6 Fault-Handling Service .............................................. 260
# R3TOS Main CPU and Application Programming Interface

8 R3TOS Main CPU and Application Programming Interface 265

8.1 The R3TOS Main CPU 266

8.1.1 External Memory Access Arbiter 267

8.1.2 HWuK-SWuK Interface 268

8.2 The R3TOS Software Microkernel (SWuK) 269

8.2.1 Management of Hardware Tasks in SWuK 271

8.2.2 High-Level Services offered by SWuK API 272

8.3 Chapter Conclusion 274

# R3TOS Prototyping and Application Development

9 R3TOS Prototyping and Application Development 275

9.1 Proof-of-Concept R3TOS Prototype 275

9.1.1 Implementation Results and Consumed FPGA Resources 278

9.1.2 Performance Results 278

9.2 Control of the Power Chain of a Railway Traction Vehicle 282

9.2.1 Application Description 282

9.2.2 Application Development 284

9.2.3 Application Functioning 289

9.2.4 Experimental Evaluation 295

9.3 SDR Application 296

9.3.1 Application Description and Development 298

9.3.2 Application Functioning and Experimental Evaluation 307

9.4 Chapter Conclusion 309

# Conclusions and Future Work

10 Conclusions and Future Work 311

10.1 Summary 312

10.2 Reflections 314

10.3 A Final Critique of R3TOS 319

10.4 Upcoming Technologies and Future Perspectives 321

10.5 Future Work 322

# Appendices

A Design and Implementation of a Fault-Tolerant PicoBlaze 327

A.1 Background: Standard PicoBlaze and ECC-Protected BRAMs 329

A.2 The ECC Processor Adapter (EPA) 330

A.2.1 JUMP Instruction 333

A.2.2 CALL and RETURN Instructions 334

A.2.3 INTERRUPT and RETURNI Instructions 335

A.3 Development Framework 337

# Bibliography

Bibliography 339
## List of Figures

1.1 Retrospective and current perspectives in computing .................................. 4
2.1 Configuration of FPGAs ............................................................................ 13
2.2 Implementation flow using Xilinx design tools ........................................ 17
2.3 ALU: HDL synthesis and netlist mapping to Xilinx Virtex-4 FPGAs .......... 19
2.4 ALU: Placement and routing in a Xilinx Virtex-4 XC4VFX12 device .......... 20
2.5 Generic structure of FPGA-based digital circuits .................................... 21
2.6 Inter-module communication infrastructures ........................................... 23
2.7 Dynamic Partial Reconfiguration ............................................................. 23
2.8 Accessing XC6200 logic cells through the FastMAP interface ............... 25
2.9 Virtex-II architecture ............................................................................. 26
2.10 General view of Virtex-4 architecture (XC4VFX12 part) ....................... 28
2.11 Slices in a CLB .................................................................................... 31
2.12 Virtex-4 SliceM .................................................................................... 32
2.13 BRAM hardware structure .................................................................... 33
2.14 Virtex-4 programmable routing lines and interconnect pattern ............. 34
2.15 PIPs in the SB ..................................................................................... 35
2.16 Detailed view of the Virtex-4 architecture ............................................ 36
2.17 Simplified structure of the Virtex-4 clock-tree ...................................... 37
2.18 Resource, SB and CB mapping to configuration memory ...................... 40
2.19 Virtex-4 frame addressing scheme (XC4VFX12 part) ......................... 40
2.20 Internal organisation of a Virtex-4 configuration frame ......................... 41
2.21 CLB frames mapping ........................................................................... 43
2.22 BRAM content data frames mapping .................................................... 44
2.23 ICAP functioning time diagram ............................................................ 51
2.24 The “bathtub” curve ............................................................................ 51
2.25 Single Event Upset (SEU) ...................................................................... 52
2.26 Standard 6-transistor SRAM cell ............................................................ 53
2.27 Radiation sensitive bits in Xilinx FPGAs .............................................. 53
2.28 Virtex-4 Frame_ECC logic functioning time diagram .......................... 56
5.1 Dittmann and Frank’s RC scenario vs. R3TOS RC scenario .............. 147
5.2 Preemptive vs. non-preemptive EDF scheduling ......................... 148
5.3 EDF and FAEDF scheduling ........................................... 151
5.4 Greatest empty rectangles in up-right, up-left, down-right and down-left directions ...................................................... 156
5.5 EAD structure .............................................................. 162
5.6 Allocating $\theta$ at candidate positions A and B ........................ 165
5.7 EAC and EVC heuristics computation .................................. 166
5.8 Allocation of tasks running at different clock rates ...................... 168
5.9 Different implementation versions of the hardware tasks .............. 169
5.10 Snake task allocation strategy ......................................... 170
5.11 Reusing partial results with Snake .................................... 171
5.12 Left half part of the simulated XC4VLX160 part ....................... 174
5.13 Performance with no damage in the FPGA device ..................... 177
5.14 Simulation results when low utilisation of the FPGA and loose real-time constraints ....................................................... 180
5.15 Simulation results when medium utilisation of the FPGA and moderate real-time requirements .......................................... 180
5.16 Simulation results when low utilisation of the FPGA and tight real-time requirements ..................................................... 181
5.17 Simulation results when high utilisation of the FPGA and loose real-time requirements .................................................... 181
5.18 Target allocations when using DRTs .................................... 183
5.19 Performance on a realistic RC simulation scenario when using the Snake approach ......................................................... 184

6.1 BUFR Diagnostic Circuit (DC) ........................................... 188
6.2 Dual-clock Management Circuit (DMC) ................................ 189
6.3 R3TOS static infrastructure ................................................ 191
6.4 Simplified generic architecture of TCL for ... ........................... 194
6.5 HWS implementation ...................................................... 194
6.6 LUT-based data buffers .................................................... 195
6.7 Resource requirement for different LUT-based buffers size .......... 196
6.8 BRAM-based data buffers ................................................... 197
6.9 Resource requirement for different BRAM-based buffer configurations .............................................................. 198
6.10 DRTs ........................................................................... 200
6.11 Representation of the BRAM switching between a computing task and a DRT .............................................................. 201
6.12 DRT implementation ........................................................ 202
6.13 Resource and configuration requirement for different DRTs ........ 204
6.14 Organisation of the Task BRAM ......................................... 206
6.15 Task state transitions ....................................................... 208
6.16 Time management in the R3TOS scheduler ............................ 210
6.17 Organisation of the FPGA State BRAM .................................. 216
6.18 EAD updating process ......................................................... 218
6.19 Simplified structure of the Architecture Checker (AC) ............... 220
6.20 Resource requirement of the Architecture Checker for different FPGA sizes ......................................................... 220
6.21 Acceleration factor achieved by the AQE for different task sizes .... 221
6.22 Simplified functioning diagram of allocator’s PicoBlaze ................ 223
6.23 Organisation of the Bitstream BRAM ........................................ 225
6.24 COSS: Configuration Operation Starting Sequence ..................... 226
6.25 Configuration template for reading-back a subset of frames from the FPGA’s configuration memory ........................................ 227
6.26 Configuration template for writing a single to the FPGA’s configuration memory or for writing the same input data to the three redundant instances of a quasi-FTU task .................................... 228
6.27 Configuration template for writing a subset of frames to the FPGA’s configuration memory ........................................ 229
6.28 Configuration template for blanking a subset of frames in the FPGA configuration memory ........................................ 230
6.29 COES: Configuration Operation Ending Sequence ..................... 231
6.30 Pad information in read-back and write operations ...................... 234
6.31 Synchronising the Frame_ECC logic and the ICAP ..................... 235
6.32 STARTUP support logic ......................................................... 236
6.33 Simplified example of communications among HWuK components when scheduling, allocating and executing a hardware task ................. 241
6.34 Master-slave communications in HWuK .................................... 242
6.35 Monitor’s state diagram ......................................................... 242
6.36 Parallelisation of R3TOS processes in the HWuK ....................... 243

7.1 Functioning of the task allocation service .................................. 246
7.2 Benefit of allocating the hardware tasks in-between clock regions ..... 248
7.3 Relocating a hardware task in-between clock regions ..................... 250
7.4 Allocating a hardware task within the footprint of a previously executed task ........................................................................ 251
7.5 Functioning of the task deallocation service ................................ 252
7.6 Task context ............................................................................ 253
7.7 Data exchange between buffers implemented using different FPGA resources ........................................................................ 257
7.8 Clock feeding to a hardware task using the BUFRs ....................... 259
7.9 Replacing damaged BUFRs ...................................................... 261
7.10 Fault-handling strategy in R3TOS ............................................. 262

8.1 Main CPU: block diagram ....................................................... 267
8.2 Memory access arbiter’s state diagram ...................................... 268
8.3 Simplified example of communications between SWuK and HWuK when executing a hardware task ........................................... 270
8.4 Management of a hardware task in SWuK .................. 272
9.1 Xilinx Virtex-4 LX development kit populated with an XC4VLX160 device .............................................................. 276
9.2 FPGA implementation of R3TOS proof-of-concept prototype ...... 277
9.3 R3TOS-based traction controller ................................. 285
9.4 PC-PWM circuit ...................................................... 287
9.5 Relation among principal periods of time in the traction controller .. 290
9.6 ICAP phase ........................................................... 291
9.7 Ping-pong functioning of PWM sample chunks .................. 292
9.8 Observed failure rate during the fault injection campaigns ....... 296
9.9 SDR demonstrator prototype ....................................... 299
9.10 Cognitive radio: DAG ............................................. 303
9.11 WiFi: DAG .......................................................... 304
9.12 WiMAX: DAG ..................................................... 305
9.13 UMTS: DAG ........................................................ 306
A.1 KCPSM3-FT: block diagram and interconnection in R3TOS HWuK ... 328
A.2 Timing of a standard PicoBlaze .................................. 329
A.3 ECC-protected BRAM internal structure .......................... 330
A.4 KCPSM3-FT instruction mapping on an ECC-protected BRAM ...... 331
A.5 Timing of KCPSM3-FT ............................................. 331
A.6 KCPSM3-FT implementation ..................................... 332
A.7 Scrubbing a single error in the program memory .................. 334
A.8 JUMP instruction timing ........................................... 334
A.9 CALL and RETURN instructions timing in PicoBlaze ......... 335
A.10 CALL and RETURN instructions timing in KCPSM3-FT ........ 336
A.11 INTERRUPT and RETURNI operation timing in PicoBlaze ....... 336
A.12 INTERRUPT and RETURNI operation timing in KCPSM3-FT ...... 337
A.13 Development framework of KCPSM3-FT ......................... 338
List of Tables

2.1 Logic resources in one CLB ............................................. 30
2.2 Frame address .............................................................. 41
2.3 BUFR frequency selection ............................................ 46
2.4 Clock source selection for the regional clock lines (RCMUX) .... 46
2.5 PIPs to route the regional clock signals to the resources in top half configuration frames ........................................ 47
2.6 Most important configuration registers .............................. 48
2.7 Most important configuration command codes ..................... 48
2.8 Packet Type1 header format .......................................... 48
2.9 Packet Type2 header format .......................................... 49
2.10 Frame_ECC Polynomial ................................................. 55

3.1 Comparison of hardware and software task compile-time characteristics ................................................................. 90
3.2 Comparison of hardware and software task runtime functions ................................................................. 91

5.1 Task-set used in Figure 5.3 .................................................. 150

6.1 Mapping of R3TOS functionality to main HWuK components ... 186
6.2 Generic parameters in the TCL ......................................... 192
6.3 Basic functions implemented by the ICAP driver ................. 232

7.1 Mapping of the R3TOS services to functions executed in the configuration manager .............................................. 245

8.1 Consumed FPGA resources by each R3TOS component .......... 273

9.1 Consumed FPGA resources by each R3TOS component .......... 279
9.2 Performance figures in the R3TOS prototype ....................... 280
9.3 Task-set in the SDR prototype ......................................... 301

A.1 Error status in ECC-protected BRAMs ............................... 330
List of Algorithms

5.1 Schedule_EDF() .................................................... 149
5.2 Schedule_FAEDF() .................................................. 153
5.3 Compute_RAM() and Compute_LAM() ............................... 156
5.4 Compute_URAM(), Compute_ULAM(), Compute_DRAM() and
    Compute_DLAM() ...................................................... 158
5.5 Compute_2DAM() ..................................................... 158
5.6 Compute_TAM() ....................................................... 160
5.7 Compute_3DAM() ....................................................... 161
5.8 Allocate_EAC() and Allocate_EVC() ................................. 164
### List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Architecture Checker</td>
</tr>
<tr>
<td>ACT</td>
<td>Application Completion Time</td>
</tr>
<tr>
<td>AET</td>
<td>Algorithm Execution Time</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AQE</td>
<td>Allocation Quality Evaluator</td>
</tr>
<tr>
<td>ARM</td>
<td>Acorn RISC Machine</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuits</td>
</tr>
<tr>
<td>BIST</td>
<td>Built In Self Test</td>
</tr>
<tr>
<td>BIF</td>
<td>Blank Frame(s) operation</td>
</tr>
<tr>
<td>BM</td>
<td>Bus Macro</td>
</tr>
<tr>
<td>BRAM</td>
<td>Block of Random Access Memory</td>
</tr>
<tr>
<td>BUFGCTRL</td>
<td>Global clock buffer</td>
</tr>
<tr>
<td>BUFBR</td>
<td>Regional clock buffer</td>
</tr>
<tr>
<td>CAF</td>
<td>Construcciones y Auxiliar de Ferrocarriles</td>
</tr>
<tr>
<td>CB</td>
<td>Connection Box</td>
</tr>
<tr>
<td>CC</td>
<td>Computation Cube</td>
</tr>
<tr>
<td>CEO</td>
<td>Chief Executive Officer</td>
</tr>
<tr>
<td>CG</td>
<td>Configuration Guardian</td>
</tr>
<tr>
<td>CIF</td>
<td>Common Intermediate Format</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computers</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block</td>
</tr>
<tr>
<td>COES</td>
<td>Configuration Operation Ending Sequence</td>
</tr>
<tr>
<td>COSS</td>
<td>Configuration Operation Starting Sequence</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>CV</td>
<td>Computation Volume</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DCM</td>
<td>Digital Clock Manager</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>DCS</td>
<td>Dynamic Circuit Switching</td>
</tr>
<tr>
<td>DLAM</td>
<td>Down-Left Adjacency Matrix</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DPR</td>
<td>Dynamic Partial Reconfiguration</td>
</tr>
<tr>
<td>DRAM</td>
<td>Down-Right Adjacency Matrix</td>
</tr>
<tr>
<td>DRE</td>
<td>Data Relocation Enable</td>
</tr>
<tr>
<td>DRT</td>
<td>Data Relocating Task</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EA</td>
<td>Evolutionary Algorithm</td>
</tr>
<tr>
<td>EAC</td>
<td>Empty Area Compaction</td>
</tr>
<tr>
<td>EAD</td>
<td>Empty Area Descriptor</td>
</tr>
<tr>
<td>EADU</td>
<td>Empty Area Descriptor Updater</td>
</tr>
<tr>
<td>EAPR</td>
<td>Early Access Partial Reconfiguration</td>
</tr>
<tr>
<td>ECC</td>
<td>Error Correction Codes</td>
</tr>
<tr>
<td>ECV</td>
<td>Exploited Computation Volume</td>
</tr>
<tr>
<td>EDF</td>
<td>Earliest Deadline First</td>
</tr>
<tr>
<td>EM</td>
<td>Electromigration</td>
</tr>
<tr>
<td>EPA</td>
<td>ECC Processor Adapter</td>
</tr>
<tr>
<td>EVC</td>
<td>Empty Volume Compaction</td>
</tr>
<tr>
<td>FAEEDF</td>
<td>Finishing Aware Earliest Deadline First</td>
</tr>
<tr>
<td>FCCR</td>
<td>Fault Containment Computation Region</td>
</tr>
<tr>
<td>FCU</td>
<td>Fault Containment Unit</td>
</tr>
<tr>
<td>FEC</td>
<td>Forward Error Correction</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FIT</td>
<td>Failure In Time</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSL</td>
<td>Full Slot Load</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>FTU</td>
<td>Fault Tolerant Unit</td>
</tr>
<tr>
<td>GCD</td>
<td>Greatest Common Divisor</td>
</tr>
<tr>
<td>GRM</td>
<td>General Routing Matrix</td>
</tr>
<tr>
<td>GSFC</td>
<td>NASA's Goddard Space Flight Center</td>
</tr>
<tr>
<td>GSR</td>
<td>Global Set/Reset</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Abstraction Layer</td>
</tr>
<tr>
<td>HBC</td>
<td>High Bandwidth Communication</td>
</tr>
<tr>
<td>HCI</td>
<td>Hot Carrier Injection</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HWS</td>
<td>Hardware Semaphore</td>
</tr>
<tr>
<td>HWuK</td>
<td>Hardware MicroKernel</td>
</tr>
<tr>
<td>ICAP</td>
<td>Internal Configuration Access Port</td>
</tr>
<tr>
<td>IDB</td>
<td>Input Data Buffer</td>
</tr>
<tr>
<td>IDF</td>
<td>Xilinx Isolation Design Flow</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>IFFT</td>
<td>Inverse Fast Fourier Transform</td>
</tr>
<tr>
<td>IGBT</td>
<td>Insulated Gate Bipolar Transistor</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite Impulse Response</td>
</tr>
<tr>
<td>IOB</td>
<td>Input Output Block</td>
</tr>
<tr>
<td>ITRS</td>
<td>International Technology Roadmap for Semiconductors</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>IVT</td>
<td>Xilinx Isolation Verification Tool</td>
</tr>
<tr>
<td>JHDL</td>
<td>Java Hardware Description Language</td>
</tr>
<tr>
<td>JIT</td>
<td>Just In Time (Synthesis)</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>KAMER</td>
<td>Keep All Maximal Empty Rectangles</td>
</tr>
<tr>
<td>LAM</td>
<td>Left Adjacency Matrix</td>
</tr>
<tr>
<td>LBC</td>
<td>Low Bandwidth Communication</td>
</tr>
<tr>
<td>LKM</td>
<td>Loadable Kernel Module</td>
</tr>
<tr>
<td>LUT</td>
<td>Look Up Table</td>
</tr>
<tr>
<td>MDR</td>
<td>Merge Dynamic Reconfiguration</td>
</tr>
<tr>
<td>MER</td>
<td>Maximal Empty Rectangle</td>
</tr>
<tr>
<td>MIT</td>
<td>Massachusetts Institute of Technology</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MTTD</td>
<td>Mean Time To Detect</td>
</tr>
<tr>
<td>MTTM</td>
<td>Mean Time To Manifest</td>
</tr>
<tr>
<td>NASA</td>
<td>National Aeronautics and Space Administration</td>
</tr>
<tr>
<td>NBTI</td>
<td>Negative Bias Temperature Instability</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>NCO</td>
<td>Numerically Controlled Oscillator</td>
</tr>
<tr>
<td>ODB</td>
<td>Output Data Buffer</td>
</tr>
<tr>
<td>OFDM</td>
<td>Orthogonal Frequency Division Multiplexing</td>
</tr>
<tr>
<td>OPB</td>
<td>On chip Peripheral Bus</td>
</tr>
<tr>
<td>OQPSK</td>
<td>Offset Quadrature Phase Shift Keying</td>
</tr>
<tr>
<td>ORGA</td>
<td>Optically Reconfigurable Gate Array</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PAR</td>
<td>Place and Route</td>
</tr>
<tr>
<td>PBR</td>
<td>Relocate a partial bitstream operation</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PC-PWM</td>
<td>Proportional Control Pulse Width Modulator</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional Integral Derivative</td>
</tr>
<tr>
<td>PIP</td>
<td>Programmable Interconnection Point</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface</td>
</tr>
</tbody>
</table>
PPC405  PowerPC 405
PRR    Partially Reconfigurable Region
PSD    Power Spectral Density
PWM    Pulse Width Modulation
QAM    Quadrature Amplitude Modulation
QoS    Quality of Service
QPSK   Quadrature Phase-Shift Keying
RAM    Random Access Memory
RAM    Right Adjacency Matrix
RBF    Read-Back Frame(s) operation
RC     Reconfigurable Computing
RCMUX  Regional Clock Multiplexer
RF     Radio Frequency
RISC   Reduced Instruction Set Computers
RM     Reconfigurable Module
RMB    Reconfigurable Multiple Bus
RMS    Root Mean Square
RMW    Read Modify Write-back
ROM    Read Only Memory
ROS    Reconfigurable Operating System
RTL    Register Transfer Level
RUT    Region Under Test
R3TOS  Reliable Reconfigurable Real-Time Operating System
SB     Switch Box
SDK    Xilinx Software Development Kit
SDR    Software Defined Radio
SEC-DED Single Error Correction Double Error Detection
SEFI   Single Event Functional Interrupt
SEU    Single Event Upset
SF     Software Function
SIMD   Single Instruction Multiple Data
SLIG   Edinburgh University's System Level Integration Research Group
SNR    Signal to Noise Ratio
SoC    System on Chip
SR     Static Region
SRAM   Static Random Access Memory
SWuK   Software MicroKernel
TAM    Temporal Area Matrix
TCL    Task Control Logic
TDDDB  Time Dependent Dielectric Breakdown
TLM    Transaction Level Models
TMR    Triple Modular Redundancy
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TUT</td>
<td>Task Under Test</td>
</tr>
<tr>
<td>ULAM</td>
<td>Up-Left Adjacency Matrix</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunications System</td>
</tr>
<tr>
<td>URAM</td>
<td>Up-Right Adjacency Matrix</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very High Speed Integrated Circuit Hardware Description Language</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VRC</td>
<td>Virtual Reconfigurable Circuit</td>
</tr>
<tr>
<td>WCDMA</td>
<td>Wideband Code Division Multiple Access</td>
</tr>
<tr>
<td>WIG</td>
<td>Wheel Impulse Generator</td>
</tr>
<tr>
<td>WiMAX</td>
<td>Worldwide Interoperability for Microwave Access</td>
</tr>
<tr>
<td>WMF2S</td>
<td>Write Multiple Frames to a Single location operation</td>
</tr>
<tr>
<td>WoD</td>
<td>Wires on Demand</td>
</tr>
<tr>
<td>WSF2M</td>
<td>Write a Single Frame to Multiple locations operation</td>
</tr>
<tr>
<td>WSF2S</td>
<td>Write a Single Frame to a Single location operation</td>
</tr>
<tr>
<td>XDL</td>
<td>Xilinx Design Language</td>
</tr>
<tr>
<td>XPS</td>
<td>Xilinx Platform Studio</td>
</tr>
<tr>
<td>XST</td>
<td>Xilinx Synthesiser</td>
</tr>
<tr>
<td>2DAM</td>
<td>2-Dimensional Adjacency Matrix</td>
</tr>
<tr>
<td>3DAM</td>
<td>3-Dimensional Adjacency Matrix</td>
</tr>
<tr>
<td>3PLD</td>
<td>3-Dimensional Programmable Logic Devices</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

“... I was sitting in the rooms of the Analytical Society, at Cambridge, my head leaning forward on the table in a kind of dreamy mood, with a table of logarithms lying open before me. Another member, coming into the room, and seeing me half asleep, called out, “Well, Babbage, what are you dreaming about?” to which I replied “I am thinking that all these tables” (pointing to the logarithms) “might be calculated by machinery.”

—Charles Babbage

The instructions to make the first modern digital computers operate were wired into the machines themselves. Practitioners quickly suffered from the lack of flexibility of fixed hardware design and came up with the stored program architecture or von Neumann architecture [Burks et al., 1989], where a stream of simple software instructions, stored in the program memory, sequentially customised the data-path of a general purpose hardware processor in order to eventually complete any complex computation. Hence, flexibility was the main motivation for the creation of software.

However, the gained flexibility was opposed to the efficiency loss due to the interactions between the program code and the underlying processor [Tiwari et al., 1994]. Indeed, the instruction fetch-decode-execute cycle is often called the “von Neumann bottleneck”, as the necessity for fetching every instruction from the program memory slows the computation [Backus, 1978]. Providing a cache between the processor and the program memory, using separate memories for data and instructions to be simultaneously accessed (i.e.,
Harvard architecture), pipelining instruction fetch-decode-execute cycle, and predicting branches are four of the ways proposed for overcoming the von Neumann bottleneck. However, these techniques only alleviate a non-solvable problem coming from the very heart of the computation model itself, and at the price of notably increasing the power consumption [Tiwari et al., 1994, Patterson and Hennessy, 2008]. It is well known Craig Bruce’s quote: “it is hardware that makes a machine fast and it is software that makes a fast machine slow”.

For more than three decades the increased integration capabilities provided by successive generations of semiconductor manufacturing were used to produce faster and more capable processors, masking the low efficiency of software-based computation [Wirth, 1995]. During this time, increases in the usable clock frequency, which was a consequence of the technology shrinkage, served as the substrate for keeping the complexity of the programs growing. Programmers simply assumed that the processors would be able to deal with the huge amount of computation their applications demanded, and thus concentrated in increasing the productivity. However, this was done by providing additional software abstraction layers which lead to ever lower efficiency (e.g., middleware). As Wirth’s law states, software got slower more rapidly than hardware became faster [Wirth, 1995].

Furthermore, although overall computing capabilities were enhanced as a result of the increased usable clock frequency, memory access bandwidth grew at slower rate, thus stressing the von Neumann bottleneck [Wulf and McKee, 1995]. But it was not the von Neumann bottleneck which led processors to stagnate. By the 2000’s the clock speed was so high that the power consumed and the heat generated by the processors made it infeasible to feed and cool them [Patterson and Hennessy, 2008]. In 2004, Intel Inc. hit the high-point in how much the clock rate could be increased with the Pentium-4 Prescott architecture. Despite the Prescott was intended to scale up to 10 GHz, this never occurred due to insolvable heat problems: even when equipped with a massive heat dissipation system and running only at 3.8 GHz, the processor reached 77 Celsius and shut itself down [Sassen, 2004]. As a result, Intel Inc. had to abandon the Prescott architecture. Processors had crashed into a power wall that prevented performance from growing, making computation hungry software applications teeter [Fish, 2012].

To cope with the power wall, processor designers adopted a divide-and-conquer strategy. More but simpler cores, which are clocked at lower frequencies and powered with
lower voltages, were and still are being integrated into multi-core and many-core processors [Patterson and Hennessy, 2008]. This permits to continue taking advantage of the advanced integration capabilities, scaling the number of cores on-chip rather than clock frequency, to achieve the ever greater performance demanded by software programmers. Unfortunately, the power wall will eventually affect multi-cores too if power per core is not reduced as the number of cores on-chip increases [Huang et al., 2008, Meenderinck and Juurlink, 2009]. In the latter reference, the authors scale the Alpha 21264 multi-core to the future technology predicted by the International Technology Roadmap for Semiconductors (ITRS), concluding that in 2022 its total consumption will exceed 7.5 times the predicted power budget. A similar conclusion is presented in [Karpuzcu et al., 2009], where the authors predict that by 2018 only 25% of on-chip cores will be able to be active simultaneously.

Meanwhile, the loss of efficiency introduced by software continues to be a not-well addressed vital issue. After all, multi-cores are nothing but an attempt to mimic the inherent parallelism of hardware using software-driven sequential processors, and thus suffering from the von Neumann bottleneck in each core [Moore, 2008, Fish, 2011]. Moreover, the achievable performance depends on the ability of the programmer to partition his/her application into independent parallelisable processes. Note that this is not always trivial due to computational complexity, or even impossible due to the inner structure of the computation. If the application partition is not successful, its parallel execution on multiple cores can introduce contention for system resources (e.g., memory accesses), making it slower than when deployed on a single core.

Furthermore, with decreasing transistor dimensions and increasing power consumption and heat dissipation, reliability due to wear-out mechanisms is expected to become a significant issue [ITRS, 2011]. Different studies predict that permanent damage will plague the chips of the future, resulting in shorter and less predictable lifetimes for them [Srinivasan et al., 2003, Borkar, 2005, White and Bernstein, 2008, White and Chen, 2008]. Certainly, in a couple of decades every manufactured chip is likely to be physically unique [Wilson, 2012] (e.g., great differences in transistor channel length and threshold voltage), thus degrading in an unique and unpredictable way over time [Mehta and DeHon, 2011]. Besides, this variability might well constrain the chip manufacturing yield.
Figure 1.1 is a position graph in computing. While the number of transistors on-chip continues to increase, performance has stalled mainly due to the power wall. Fortunately, in the current state of technological development there is an opportunity to tackle this issue.

Reconfigurable hardware, such as Field-Programmable Gate Arrays (FPGAs), gets its name from its physical yet reconfigurable interconnect fabric, which can be ’molded’ at runtime to implement a variety of different specific functionalities. Therefore, reconfigurable hardware has the potential to bring a new computation scenario that crosses the decades old hardware-software boundaries, combining the flexibility of software with the speed and efficiency of pure hardware.

In Reconfigurable Computing (RC) the computation is performed by building circuits rather than executing instructions. The created circuits include specialised data-paths with the strictly necessary memory and processing components coupled in custom deep pipelines to perform each type of computation with the desired level of accuracy, i.e., using custom data bit-lengths. The data to be processed is progressively transformed into results as it flows through the data-paths and thus, as the computation is driven by its own data-stream, the von Neumann bottleneck can be circumvented. In effect, specialised data-paths offer data-level parallelism as different components of the data-path can simultaneously process different sets of the same data-stream. Besides, data-level parallelism can be combined with process-level parallelism to achieve even higher performance [Banerjee et al., 2006].
In terms of performance figures, FPGAs show an internal bandwidth to move data in the order of terabytes/s, a throughput of integer operations in the order of teraoperations/s and of floating-point operations in the order of gigaflops/s [Prasanna Sundarajan, 2010, Strenske et al., 2010]. As a result, some ten-fold improvement in the execution speed of many popular applications can be achieved, even at low clock frequencies [Compton and Hauck, 2002, Todman et al., 2005, Tian and Benkrid, 2010, Hameed et al., 2010, Prasanna Sundarajan, 2010, Feldman, 2011, Yan et al., 2011, Tse, 2012]. More significantly, computation specialisation can improve energy efficiency by up to some hundred-fold [El-Ghazawi et al., 2008, Tian and Benkrid, 2010, Prasanna Sundarajan, 2010], allowing to deal with the power wall.

Computation specialisation is not a new concept at all. Complex Instruction Set Computers (CISCs) used to rely on specialised hardware to rapidly perform some complicated types of computation. However, when it was found that most of the complicated functions provided by hardware accelerators were rarely used in actual programs, Reduced Instruction Set Computers (RISCs) appeared [Patterson and Ditzel, 1980, Furber, 2000]. RISCs replaced the rarely used circuitry with more registers and cache memory to achieve faster execution of simpler instructions.

While most of the applications that are currently accelerated with FPGAs sacrifice flexibility (i.e., they target single algorithms with fixed and dedicated implementations), we believe that this capability is to play a key role in the future. As the amount of on-chip hardware that can be active at any particular time is getting limited by the power wall, the \textit{online specialisation} offered by reconfigurable hardware emerges as a promising way to combine computation in space and time to obtain the best performance per transistor count and unit of consumed energy; i.e., the same on-chip resources can be reused efficiently to implement different functionalities over time. Note that this differs from CISCs as the circuitry to implement each specific functionality occupies space on the chip and consumes power only when it is required by the program. Besides, online specialisation would allow to accelerate algorithms that are too unpredictable and dynamic to be implemented as fixed designs, but that include some computation-intensive repetitive parts which could individually take advantage of an FPGA implementation. All in all, reconfigurable hardware seems to be appropriate to implement on-demand hot-pluggable and efficient hardware accelerated func-
tions for RISC processors. Indeed, the two major FPGA companies, Xilinx Inc. and Altera Corp., have both recently introduced new platforms that integrate an ARM Ltd. designed RISC processor within their own configurable logic: Zynq Extensible Processing Platform (Xilinx Inc.) and Altera System-on-Chip (Altera Corp.). Another big name company doing research on this line is Microsoft Inc., which has recently released a dynamically extensible RISC processor \[\text{Chen et al., 2010}\].

The same flexibility that makes online specialisation possible also permits to solve, or at least mitigate, the reliability concerns that appear when pushing semiconductor manufacturing to its physical limits. In effect, reconfigurable hardware allows the building of autonomous systems capable of keeping their architecture fault-free by reconfiguring around damaged portions when needed, thus minimising the impact of aging on performance and maintenance costs. Dr. Peter Cheung, head of the Department of Electrical and Electronics Engineering in Imperial College London, goes further in this line and suggests that “in a few decades, every system-level device will have to be programmable to configure its still-functioning resources into a working system” \[\text{Wilson, 2012}\]. Besides, such a reconfigurable system could adapt itself after having been manufactured to the specific physical features of the chip where it is implemented (i.e., variation in transistor characteristics), thus increasing the chip fabrication yield.

Despite the numerous advantages reconfigurable hardware is to bring, its success is highly conditioned by the way it reaches application developers. After decades of prevalence, the software programming style has spread through all application domains, including even those which were traditionally hardware-centered, and thus, it seems currently impossible to radically change this situation without a major collapse in productivity and increase in cost. Faced with this, the concept of a \textbf{Reconfigurable Operating System (ROS)} to give to reconfigurable hardware a “software look and feel” has been gaining momentum since the mid-90’s \[\text{Brebner, 1996, Wigley and Kearney, 2001, Walder, 2005}\]. The long-term objective is to make hardware-related benefits exploitable by traditional software-centric programmers, freeing them from knowing the low-level details.

At design time, the ROS would ease the hardware-software codesign, isolating the low-level hardware architectures from the high-level software services which run upon them, but at the same time offering a common infrastructure to give support and cohesion to the
final integrated system. This is in line with the current trend of high-level synthesis, which permits the building of Register-Transfer Level (RTL) designs starting from the behavioural specifications in an automatic manner [Cong et al., 2011]. In fact, the use of a ROS would be an excellent way to deal with design complexity and productivity, allowing for a sustainable growth of the applications by leveraging the amount of software and hardware on them. Furthermore, the ROS would promote the reusability of tested functionality, speeding up development cycles and shortening time-to-market, and once the product has been launched, it could allow for simpler and easier upgrades to prolong its life-cycle in the market.

At runtime, the ROS would grant the system some measure of autonomy over its own resources to carry out self-healing and to schedule the computation, swapping in and out the specialised data-paths on behalf of the user. As a result the user will have a large and reliable virtual hardware resource implemented using limited and unreliable physical reconfigurable hardware [Singh and Bellec, 1994], which is spatially and temporally managed by the ROS. Moreover, a ROS could offer outstanding real-time performance as hardware-based computation leads to simpler timing behaviour, which can be usually predicted with a precision of nanoseconds.

On the other hand, the ROS introduces some performance and consumption penalties due to the silicon overhead required by its own logic, and due to the time overhead needed to execute its own background processes. Another disadvantage of using a ROS is the existing possibility that a fault affecting it makes the complete system fail. These three metrics (i.e., area requirement, timing behaviour and robustness against faults) are the main indicators of the quality of a ROS. Nevertheless, despite the existence of these limitations, major processor manufacturers have included in their roadmap the necessity for adding to their ever more complex computing devices (e.g., multi-cores) some built-in logic to support OS-like services [Intel Inc., 2005].

The central aim of this thesis is to explore the feasibility of developing a fully operational ROS, acknowledging the limitations of current reconfigurable technology, namely of Xilinx partially reconfigurable FPGAs. The objective is not to solve a particular problem, but to develop a cross-domain technology that could be used in a wide range of applications with different requirements of performance, fault-tolerance, and real-time. Indeed, the thesis is aimed to replace some of the supporting software layers which currently run beneath user's
applications with more efficient hardware. Of course, to guarantee the transparency to the application developer, the services provided by the underlying supporting hardware should be called in the same way as is traditionally done with the software layers.

This research ranges from foundations (e.g., algorithms to schedule the computation and to manage the reconfigurable resources under efficiency and dependability premises), to practical implementations that demonstrate these concepts by means of two case-studies: (1) dependability-oriented control of the power chain of a railway traction vehicle and (2) data-streaming oriented Software Defined Radio (SDR).

1.1 Open Problem

Despite the clear potential of partially reconfigurable FPGAs, to push the current power wall beyond what is possible with general purpose processors, as well as to meet ever more exigent reliability requirements, the lack of standard tools and interfaces to develop reconfigurable applications limits FPGAs user base, and makes their programming even less productive. This is a vicious circle that requires intervention to prevent it from continuing.

1.2 Objectives of the Thesis

The main objective of the thesis is to come up with a solution for the aforementioned problem. It can be summed up as follows:

“To propose and develop a way to universalise the exploitation of the advanced reconfiguration capabilities of modern FPGAs towards more advanced and sophisticated applications and ultimately, towards progress”

This long-term objective can be divided into four sub-objectives:

- **Real-Time**: Develop the necessary support for exploiting the inner predictability of pure hardware to achieve real-time performance.

- **Dependability**: Develop the necessary support for exploiting the flexibility of FPGAs to build systems able to configure their own resources, maintaining their functionality in the presence of permanent defects and spontaneous faults.
• **High-Performance**: Develop the necessary support for exploiting the flexibility of FP-GAs to load specialised circuits upon demand, each performing a specific type of computation in an efficient way.

• **High-Level Programming**: Provide the means to make the aforementioned capabilities easy-to-use without requiring any knowledge of the low-level FPGA details.

### 1.3 Achievements of the Thesis

This thesis outlines the first **Reliable Reconfigurable Real-Time Operating System (R3TOS)**, which is being developed in collaboration with other members in the System Level Integration Research Laboratory (SLIG) of the University of Edinburgh. The website of R3TOS can be visited at http://www.see.ed.ac.uk/~kbenkrid/R3TOS. R3TOS integrates the most important achievements of the thesis, which are next enumerated:

1. **Theory**:
   - The definition of novel computing and dependability models for RC [Iturbe et al., 2013b].

2. **Algorithms**:
   - The definition and implementation of a novel scheduling algorithm (FAEDF), two novel allocation heuristics (EAC and EVC) and a task placement strategy (Snake), specifically conceived to be used in R3TOS [Iturbe et al., 2011a, Iturbe et al., 2011d, Iturbe et al., 2013a].

3. **Implementation**:
   - The development of a fully operational 2-Dimensional relocation solution, including support for placing the hardware tasks in-between clock region boundaries.
   - The development of a novel management of FPGA clocking resources in order to run each hardware task at its maximum clock frequency as well as to recover from failed clock sources [Iturbe et al., 2012].
• The development of a set of mechanisms to support hardware multitasking in FPGAs, namely for inter-task communication and synchronisation [Iturbe et al., 2011c].

• The development of a novel fault-handling strategy, which increases the system availability and, differently from other mitigation strategies, addresses both transitory upsets and permanent damage to the reconfigurable device.

• The development of a seamless execution environment where both software and hardware tasks can cooperate together.

1.4 Outline of the Thesis

This thesis is structured in ten chapters and one appendix:

Chapter 1 - Introduction: The chapter you are now reading.

Chapter 2 - Introduction to FPGAs and Dynamic Partial Reconfiguration:

“What is it all about?”

Reviews digital design and dynamically and partially reconfigurable Xilinx FPGAs.

Chapter 3 - State of the Art of Operating System Support for FPGAs:

“What is the starting point?”

Presents related state of the art on Reconfigurable Operating Systems (ROS) for FPGAs, highlighting the major concepts and issues.

Chapter 4 - Overview of R3TOS and associated Computing and Reliability Models:

“How does it work?”

Introduces the general architecture of R3TOS and its major capabilities.

Chapter 5 - R3TOS Algorithms to Manage Workload and FPGA Resources: “How does it think?”

Describes the scheduling and allocation algorithms used in the R3TOS microkernel.

Chapter 6 - Low-Level Hardware Support for the R3TOS Microkernel:

“What does it look like?”

Describes the architecture of the R3TOS microkernel.
Chapter 7 - Low-Level R3TOS Microkernel Services: "What does it feel like?"

Describes the low-level support offered by the R3TOS microkernel to interact with the underlying FPGA hardware.

Chapter 8 - The R3TOS Main CPU and API: "How is it used?"

Describes the high-level support offered by R3TOS to the final user to develop his/her application.

Chapter 9 - R3TOS Prototyping and Application Development: "What can it do?"

Presents a set of performance and reliability results measured when using R3TOS in two different case-studies: traction control of a railway vehicle and multi-standard SDR.

Chapter 10 - Conclusion and Future Work: "What is the bottom line?"

Presents the conclusions and points out future work.

Appendix A - Design and Implementation of a Fault-Tolerant PicoBlaze (KCPSM3-FT):

Discusses how to extend a standard Xilinx PicoBlaze 8-bit processor to execute its program from an ECC-protected BRAM.
Chapter 2

Introduction to FPGAs and Dynamic Partial Reconfiguration

Since the invention of the first transistor in 1947 to our days, digital systems have experienced a continuous and relentless development. This innovation revolutionised electronic computers in terms of speed, mass and power consumption [Brinkman et al., 1997]. 1958 was the year of birth of the first integrated circuit, which included several transistors in a single piece of semiconductor. The efforts in the following years focussed on increasing transistor integration capacity, giving rise to Application Specific Integrated Circuits (ASICs) [Smith and Sebastian, 1997]. The next step was to put a computer-in-a-chip, which finally came to light in the early 70’s [Patterson and Hennessy, 2008]. Microprocessors started a new trend for programmability, which eventually led to the creation of Field-Programmable Gate Arrays (FPGAs) in the early 80’s [Carter, 1997].

Unlike fixed ASIC, the physical resources of an FPGA chip can be appropriately configured to create the desired system after manufacturing and, unlike a microprocessor, the created system does not incur low efficiency. Likewise, while ASICs simply tolerate faults by using redundancy, FPGAs permit to route around damaged resources on-the-fly.

Configurability is achieved by introducing an additional layer on top of the fixed circuitry, which typically consists of a configuration memory that stores the information defining the connection and operation of the underlying physical resources (see Figure 2.1).
FPGAs have continuously evolved and now include up to tens of thousands of configurable logic blocks as well as hundreds of memories and signal processing blocks, allowing to build very capable Systems-on-Chips (SoCs) [Hauck and DeHon, 2007]. The computation performance delivered by FPGAs (i.e., clock frequency increase x logic resource count increase) has improved approximately by 92x over the past decade, while their cost has decreased by 90% in the same period [Prasanna Sundarajan, 2010]. As a result, FPGAs are progressively gaining importance in the current competitive electronic market. The prediction that the FPGA market will hit $6bn by the end of 2015 made by the CEO of Xilinx Inc., the major FPGA vendor, supports this statement [Manners, 2010].

However, the way has not always been easy. During the early years, FPGAs were consigned to play a secondary role due to the low integration capabilities; e.g., as a coprocessor of a CPU. But precisely, the fact of growing in a market vastly dominated by microprocessors led to a very important milestone in 1997. In that year the first dynamically and partially reconfigurable FPGA family, the XC6200, was commercialised by Xilinx Inc. [Xilinx Inc., 1997]. The XC6200 was inspired by research work previously conducted at the University of Edinburgh by Dr. Kean [Kean, 1988], who indeed was contracted by Xilinx Inc. to take part in its development. Based on the fact that processors run different programs at different times, Dr. Kean thought that FPGA-based coprocessors should be able to be reconfigured for implementing different functions at different times as well. This was the main motivation for making all the logic and routing resources dynamically reprogrammable in the XC6200. Furthermore, the configuration memory of the XC6200 was accessible through an interface designed to connect directly to the host microprocessor, allowing it to perform simple hardware configurations and fast data transfers between the microprocessor and the FPGA [Churcher et al., 1995, Xilinx Inc., 1996a, Xilinx Inc., 1996b].
This chapter is intended to prepare a reader who is not familiar with FPGA technology to better understand the work carried out in this thesis. Namely, it presents the architecture, capabilities and limitations of currently available partially reconfigurable FPGAs. It describes the tools and methods to implement a dynamically reconfigurable system using FPGAs and introduces the major reliability concerns in these devices. The chapter also reviews the most significant related work.

2.1 Digital Design and Implementation

Digital design has developed in accordance with the growth of electronic devices. While in the early days the designer was focussed on the gate level, as technology evolved Hardware Description Languages (HDLs), such as VHDL or Verilog, appeared to ease the design and verification of more complex circuits. These languages were intended to offer a text-entry replacement to traditionally used electronic schematics, but still preserving the main ‘hardware’ features of the latter (e.g., signals or ports) and including constructs to handle parallelism (e.g., processes). Among the main advantages of HDLs are improved comprehensibility, portability, reusability and hierarchical design.

Later, with the objective of increasing productivity and universalising digital design, even among non-hardware-oriented developers, object-oriented software was provided with support to deal with hardware matters [Cong et al., 2011]. For instance, SystemC extends C++ [Coussy and Morawiec, 2008] and JHDL extends Java [Bellows and Hutchings, 1998]). In fact, some properties existing in software are interesting when dealing with digital design. For instance, software inheritance is a good way to promote hardware hierarchical design and reusability, and class constructors/destructors are an interesting way to control the reconfiguration in partially reconfigurable systems. In this line, especially promising are the Kiwi project, which is aimed at synthesising circuits from high-level models written using regular multi-threaded software languages [Singh and Greaves, 2008], and AutoESL’s AutoPilot, which is a high-level synthesis tool whose results are reported to match or exceed hand-coded RTL for data path-intensive and DSP designs [Berkeley Design Technology Inc., 2010]. In fact, the latter has been recently acquired by Xilinx Inc. and included in its new Vivado Design Suite [Xilinx Inc., 2012c]. AutoPilot has also been used to develop a CUDA-
Chapter 2- Introduction to FPGAs and Dynamic Partial Reconfiguration

to-FPGA design flow (i.e., FCUDA [Papakonstantinou et al., 2009]), where a CUDA program is first translated into multi-threaded C and, finally synthesised to RTL.

Common in HDLs is the high-level of abstraction. The designer specifies the behaviour of the circuit he/she wants to create (e.g., using $+$, $-$, $\&$, $\star$, $\star\star$, $/$, mod, abs, rem, $>$, $<$, $\neq$, $=$, $\geq$, $\leq$ operators, repetitive loops, conditional statements, etc.) and forgets about dealing with the detailed gate-level implementation. This is carried out by the Synthesiser, Mapper, Placer and Router in four main stages [Hauck and DeHon, 2007].

1) Synthesis of the HDL behavioural description into an equivalent circuit that uses generic hardware primitives (e.g., adders, multipliers, counters, logic gates, etc.). The resulting technology-independent RTL circuit is named as Netlist. This process is comparable to the compilation of high-level software languages into assembler instructions [Gajski et al., 1992]. For instance, Figure 2.3a depicts the inferred netlist from the VHDL description of the ALU shown in Listing 2.1. This netlist includes only two primitives: an adder/subtractor and an 8-input NOR gate.

2) Technology Mapping of the netlist to the logic resources of the target architecture. When the circuit is to be implemented as an ASIC, the chosen hardware primitives are later transformed into specific geometries in silicon. On the other hand, when the circuit is to be implemented using an FPGA, the set of usable primitives is constrained by the built-in resources in the device. Note that the designer can directly instantiate existing hardware primitives in the target architecture in light of a more optimised circuit, and at the price of sacrificing portability to other architectures. The resulting mapped primitives make up the technology-mapped netlist. For instance, Figure 2.3b depicts the technology-mapped netlist of the previously presented ALU when targeting a Xilinx Virtex-4 FPGA: the adder/subtractor and the 8-input NOR gate primitives are mapped to a set of 3/4-input LUTs, 2-input XOR gates and multiplexers.

3) Placement of the hardware primitives included in the technology-mapped netlist to actual locations in the ASIC silicon die, or to available resources in the FPGA device. It is important to note that the designer can direct this process by using some placement constraints, like “assign this resource to this hardware primitive” or “never use this resource”. Figure 2.4a shows the placed but unrouted ALU in an XC4VFX12 part.
4) **Routing** to make the necessary connections among the placed resources in the device. When the circuit is implemented on an FPGA, routing consists of configuring the switching crossbars in the device. As the routing is highly influenced by the placement of resources, Place And Route (PAR) are usually executed together to obtain the best results (i.e., reduce the total wire length). In contrast to the placement, there are no consolidated constraints to direct the routing when using Xilinx FPGAs. Figure 2.4b depicts the fully routed ALU. Note that a placed and routed netlist can be saved as a **hard-macro** to be used in other designs. Hard-macros always retain their original features regardless of the target design in which they are used as they are not placed and routed again. Typically hard-macros are optimised for power, area or timing.

The synthesis ends with the generation of a layout mask to manufacture an ASIC, or with the generation of a **configuration bitstream** to configure an FPGA. FPGA bitstream is used to be exclusively generated with proprietary vendor tools due to its closed format. In the specific case of Xilinx, however, there is an opportunity to access a very low-level description of the FPGAs internal state using Xilinx Design Language (XDL) \[ Beckhoff et al., 2011 \]; i.e., a placed and routed design can be transformed into a readable XDL description file and vice versa. Listing 2.2, 2.3 and 2.4 show three exemplary pieces of the XDL description of the above-presented ALU implementation.

The synthesis may fail either because the circuit cannot be accommodated onto a given device (e.g., there are not sufficient amount of resources in the chip or the available silicon die is too small) or because any of the constraints imposed by the designer cannot be fulfilled (e.g., the circuit cannot work at the specified clock frequency).

Figure 2.2 depicts the implementation flow for the specific case of Xilinx FPGAS. All file extensions at the different implementation stages as well as some of the available tools are shown in this figure. For instance, .NGD is the extension for the netlist file, .NCD is for the technology-mapped netlist and .BIT for the final bitstream.

In the last years the possibility of executing the aforementioned implementation flow online has been proposed in order to create FPGA hardware on-the-fly based on the necessities of the system at any time \[ Steiner, 2008, Bergeron et al., 2007, Vahid et al., 2008 \].

\[In any case, it is possible to manually route a signal using Xilinx design tools (i.e., FPGA_Editor) and save a specific constraint for that route in light of preserving it in other implementations.\]
Vahid et al. refer to this as Just-In-Time (JIT) synthesis and have developed very light PAR algorithms able to run in on-chip embedded processors. They proved JIT synthesis in a custom device with a similar architecture to Xilinx FPGAs. Unfortunately, JIT synthesis is limited for commercial FPGAs as their bitstream format remains closed. Some research efforts have recently tried to break this wall down, leading to the creation of novel design tools which assist the high-level user accessing the bitstream of (Xilinx) FPGAs. These efforts are described in section 2.4. Alternatively, other works propose to implement a custom reconfigurable circuit (i.e., configuration memory and logic) on top of an ordinary FPGA to gain full control over each configuration bit. This is known as Virtual Reconfigurable Circuit (VRC) [Sekanina, 2003] and its main disadvantage is the significant area overhead.

### 2.1.1 FPGA-based Synchronous Design

Figure 2.5 shows a typical circuit implementation on an FPGA: the combinatorial logic is sandwiched in-between the sequential components (e.g., flip-flops), which are driven by the same clock signal. In order to ensure that clock edges arrive synchronously to all of the sequential components in the device, the clock signal is routed separately through a dedicated clock-tree. This way of implementation is called synchronous design and its main merit
Listing 2.1: ALU: VHDL behavioural description (Addition and Subtraction functions with ZERO flag)

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ALU is
port (op_A: in std_logic_vector(7 downto 0);
op_B: in std_logic_vector(7 downto 0);
opcode: in std_logic;
ALU_z: out std_logic;
ALU_out: out std_logic_vector(7 downto 0));
end ALU;

architecture behavioural of ALU is
begin
signal ALU_tmp: std_logic_vector(7 downto 0);

process(op_A, op_B, opcode)
begin
  case opcode is
  when '0' => ALU_tmp <= op_A + op_B;
  when '1' => ALU_tmp <= op_A - op_B;
  when others =>
  end case;
end process;

process(ALU_tmp)
begin
  if (ALU_tmp = "0") then
    ALU_z_flag <= '1';
  else
    ALU_z_flag <= '0';
  end if;
end process;

ALU_out <= ALU_tmp;
end behavioural;
```
Figure 2.3: ALU: HDL synthesis and netlist mapping to Xilinx Virtex-4 FPGAs
is simplified timing analysis [Friedman, 2001]. Indeed, the synthesiser uses a very precise model of the FPGA where all its hardware primitives, including wire routes, are conveniently characterised. This means that the maximum propagation delay of the combinatorial signals, the longest path, can be estimated with precision. The circuit is thus temporarily stable if its longest path is shorter than the clock period, ensuring all signals are stable at every clock edge. To meet the timing constraints of a circuit, related logic is placed adjacent to reduce the propagation delay provoked by routing. Retiming is another way to increase the maximum usable clock frequency. It consists in moving the flip-flops across combinatorial logic in the longest path to improve the timing [Cong and Wu, 1999, Singh and Brown, 2002, Singh et al., 2005].

2.1.2 Modular SoC Design

Among the benefits brought about by HDLs, the most important is reusability. Indeed, the experienced “SoC revolution” in the last two decades has been possible in part thanks to
Chapter 2- Introduction to FPGAs and Dynamic Partial Reconfiguration

**Figure 2.5: Generic structure of FPGA-based digital circuits**

**Listing 2.2:** ALU: Example of XDL description of a placed logic block (and its internal configuration to implement a 4-input NOR logic function)

```plaintext
1 inst "ALUz_cmp_eq0000_map6" "SLICEL", placed CLB_X28Y31 SLICE_X44Y62 ,
2 "BXINV::#OFF BYINV::#OFF CEINV::#OFF CLKINV::#OFF COUTUSED::#OFF
3 CY0F::#OFF CY0G::#OFF CYINIT::#OFF DXMUX::#OFF DYMUX::#OFF
4 F:ALUz_cmp_eq000012:#LUT:D=(~A1*(~A2*(~A3*~A4)))
5 F5USED::#OFF FFX::#OFF FFX_INIT_ATTR::#OFF FFX_SR_ATTR::#OFF
6 FFY::#OFF FFY_INIT_ATTR::#OFF FFY_SR_ATTR::#OFF FXMUX::#OFF
7 FXUSED::#OFF G::#OFF GYMUX::#OFF REVUSED::#OFF SYNC_ATTR::#OFF
8 XBUSED::#OFF XMUXUSED::#OFF XUSED::0 YBUSED::#OFF
9 ;
```

**Listing 2.3:** ALU: Example of XDL description of a placed I/O port (and its internal configuration)

```plaintext
1 inst "ALUz" "IOB", placed IOIS_NC_X30Y42 D3 ,
2 "DIFFI_INUSED::#OFF DIFF_TERM::#OFF DRIVEATTRBOX::12 DRIVE_0MA::#OFF
3 GTSATTRBOX::#OFF INBUFUSED::#OFF IOATTRBOX::LVCMOS25 OUSED::0
4 PULL::#OFF SLEW::SLOW TUSED::#OFF OUTBUF:ALUz_OBUF: PAD:ALUz:
5 ;
```

**Listing 2.4:** ALU: Example of XDL description of a route that connects the logic block in listing 2.2. with the I/O block in listing 2.3.

```plaintext
1 net "ALUz_cmp_eq0000_map6",
2 outpin "ALUz_cmp_eq0000_map6" X ,
3 inpin "ALUz_OBUF" F3 ,
4 pip CLB_X27Y42 IMUX_B9_INT -> F3_PINWIRE0 ,
5 pip CLB_X28Y31 X_PINWIRE0 -> BEST_LOGIC_OUT50_INT ,
6 pip INT_X27Y42 W2MID2 -> IMUX_B9 ,
7 pip INT_X28Y31 BEST_LOGIC_OUT50 -> N6BEG9 ,
8 pip INT_X28Y37 N6END9 -> N2BEG9 ,
9 pip INT_X28Y40 N2END_N9 -> N2BEG1 ,
10 ;
```

This feature [Martin and Chang, 2003]. The other reason for this revolution has been the ever greater integration capabilities.

Currently, it is possible to develop in a very short time a complete SoC by aggregating multiple modules which may have been designed by different third-party companies, perhaps not precisely for that system. This is known as modular design. HDLs permit to instantiate as many modules, and as many times as needed each of them, to form the desired SoC. Furthermore, most HDLs are parameterisable, allowing fine-tuning of the modules to the specific requirements of the system where they are being used.
Notably, modular design increases the abstraction level as the HDL files of the reusable modules belong to a low-level hierarchy with little interest for the top-level integrated system designer, who simply uses the modules in his/her specific SoC. For that purpose the designer relies on the documentation provided by the developer of the modules, describing their interfaces and configurable parameters.

In this context the most important tasks assigned to the system designer are to interconnect the modules by means of a communication infrastructure and to validate the complete system. It is normal practice to use standardised communication infrastructures that ensure compatibility and encapsulation among the modules to facilitate the composability and diagnosis of the system [Kopetz and Bauer, 2003]. Standardised communication infrastructures permit the use of Transaction Level Models (TLMs) for early verification and functional debugging [Grotker, 2002]. This is much more appropriate than low-level RTL to simulate complex SoCs.

The process of attaching a set of application-dependent modules to a communication infrastructure to build a SoC is known as platform-based design. In platform-based design the communication infrastructure itself is treated as another module and customised to the specific needs of the SoC (e.g., number of attached modules, word bit-length, etc.). For instance, the Platform Studio developed by Xilinx Inc. (XPS) uses CoreConnect buses and presents the designer with a graphical interface which simplifies the design process to almost a push-button process.

As the number of modules in the SoC increases and the FPGAs scale, Networks-on-Chip (NoC) are preferred over traditional on-chip shared buses or crossbars [Bolotin et al., 2004]. In this approach the information exchanged by the modules is formatted into data packets prior to being sent through the NoC. As can be seen in Figure 2.6c, the NoC structure consists of multiple point-to-point links interconnected by means of routers, which dispatch the data packets based on the logical address of the destination module(s) [Benini and De Micheli, 2002, Pasricha and Dutt, 2008]. As multiple links and routers can be used simultaneously, NoCs support concurrent data exchange, leading to much higher communication bandwidth than buses [Palesi et al., 2007]. Other advantages of NoCs include flexibility, scalability and reliability, e.g., alternative logic routes can be used when any of the links fails [Pirretti et al., 2004].
2.2 Dynamically and Partially Reconfigurable Xilinx FPGAs

Of particular interest to this research are FPGAs that include Dynamic Partial Reconfiguration (DPR) capability. As the name implies, DPR permits the reconfiguration as many times as needed and dynamically of the functionality implemented by a specific portion of the FPGA while the rest of the device continues to operate unchanged. Consequently, hardware resources can be time-multiplexed, leading to more efficient and powerful computing systems (see Figure 2.7). Besides, DPR permits a reduction in the reconfiguration times, bitstream storage requirements and power consumption, by limiting reconfiguration to the specific used resources.

DPR is possible in Xilinx devices because their configuration memory is SRAM-based. Indeed, Xilinx FPGAs have supported DPR since the release of the early XC6200 family in the mid 1990’s and it has been progressively enhanced in subsequent devices (i.e., Virtex-II/4/5/6/7 FPGAs). In addition, the successful use of DPR in the Spartan-3 low-cost family of FPGAs has been reported [Gonzalez et al., 2007] and the next low-cost generation family, Spartan-6, naturally supports it. It is also important to note that the second major FPGA vendor, Altera Corp., has recently put onto the market its first family of devices with support for DPR, Stratix-V [Altera Inc., 2010]. Therefore, the two major FPGA manufacturers, repre-
senting nearly 90% of the current market, agree that DPR is to play an important role in the future.

When developing this research the latest device fully supported by Xilinx DPR design tools was Virtex-4, and thus this thesis mainly uses this family of FPGAs. In any case, most of the achievements are also applicable to newer families of Xilinx FPGAs (e.g., Virtex-6 and Virtex-7), whose architecture is not fundamentally different from Virtex-4 for the purpose of this research (e.g., all of them are tile-based).

2.2.1 Evolution of Partially Reconfigurable Architectures

This section presents the evolution of Xilinx partially reconfigurable FPGAs. The objective is to establish the context in which the Virtex-4 family appeared.

The XC6200 Family

The Xilinx XC6200 family was the first commercial FPGA with support for DPR [Xilinx Inc., 1997]. Its architecture included a fine-grained regular array of logic cells, routing resources and a pioneering memory-mapped I/O interface, called FastMAP (see Figure 2.8) [Churcher et al., 1995]. FastMAP consisted of a 16-bit address bus, control signals (such as Read/Write) and a data bus configurable to different widths (8, 16 and 32 bits). A Map Register allowed mapping the logic cells in a column of the chip onto the data bus and, thus, arbitrarily located cells were collected together to form complete words of a distributed configuration memory. The column addressing of the cells was implemented by a decoding circuitry. Moreover, a Mask Register allowed the masking out of a subset of bits within a word. As a result the logic cells could be individually accessed in the configuration memory, and hence the device could be partially reconfigured.

The Virtex, Virtex-II and Virtex-II Pro Families

The Xilinx Virtex, Virtex-II and Virtex-II Pro families included improved features with regard to their predecessor. The logic cells gave rise to more versatile Configurable Logic Blocks (CLBs), which provided logic, arithmetic, data storage and data shifting functions. I/O ports were replaced by the more sophisticated I/O Blocks (IOBs) and clock management resources were included in the central part of the chip. Moreover, FPGAs moved from being homogeneous to become heterogeneous devices with some specialised resources embed-
ded in their architecture, namely 18Kb SelectRAM dual-port memory blocks and 18-bit x 18-bit multipliers. This allowed the designer to take advantage of pre-made hardware functionality to increase the performance of the designs. Gigabit transceivers and hard-wired PowerPC 405 processors were also added to Virtex-II Pro architecture. This was an important step towards fully reconfigurable SoCs with both instruction level reconfigurability (delivered by PPC405 processors) and logical level reconfigurability (delivered by reconfigurable hardware).

The FastMAP of the former XC6200 family was replaced by the SelectMAP interface. Moreover, Xilinx Virtex-II and Virtex-II Pro included a special configuration interface called the Internal Configuration Access Port (ICAP), which was a functional subset of the SelectMap accessible from inside the FPGA. Hence, the ICAP could be driven by the circuit implemented in the FPGA [Blodget et al., 2003], removing the need for an external microprocessor for controlling the reconfiguration processes, and therefore enabling the development of self-reconfigurable systems. Both SelectMAP and ICAP were 8-bit width and could operate up to 66 MHz [Xilinx Inc., 2007a].

The Virtex configuration memory was organised in vertical configuration frames that spanned the whole height of the device (see Figure 2.9). A configuration frame was the smallest amount of configuration information that could be accessed in the configuration
memory, and its length varied depending on the number of resource rows in the device. For instance, a small XC2V40 part had only 832 bits per frame while a larger XC2V1000 had 3392 bits. Note that every frame included configuration information related to IOBs, which were located around the perimeter of the FPGA (see Figure 2.9). This organisation of the configuration memory was a major problem for implementing reconfigurable systems with 2-Dimensional floor-plans. Indeed, these systems existed only in theory and real implementations were limited to only 1-Dimension.

As there was no masking register available in the Virtex architecture, it was not possible to access each resource individually within the configuration frames. Instead, DPR was based on the glitchless configuration principle, which states that when a configuration bit holds the same value before and after configuration, the resource controlled by that bit does not experience any discontinuity in operation while the reconfiguration process takes place. Hence, limited partial reconfigurability was possible using a Read-Modify-Writeback (RMW) sequence. However, RMW was unable to deal with information susceptible of being dynamically modified by the system’s own operation (e.g., data or state information). This information was inevitably modified whenever the frame where it was mapped to was rewritten and, considering the long time needed to reconfigure a single frame information, corruption was likely to occur.
2.3 Virtex-4 Architecture

The Virtex-4 family brings renewed computation and reconfiguration capabilities with regard to former Virtex families [Xilinx Inc., 2008b].

Virtex-4 XtremeDSP slices (also named as DSP48s) extend former Virtex embedded multipliers with additional pipelined stages, including an adder/subtractor and a 48-bit accumulator. Virtex SelectMAP memory blocks are replaced with improved BRAM primitives which can also be configured as FIFOs. Unlike earlier Virtex architectures, BRAMs and DSP48s are separated in different columns and IOBs are grouped together in dedicated columns, leading to the device structure depicted in Figure 2.10. Note that DSP48s, IOBs and BRAMs can be seen as the heterogeneous resources embedded in the middle of the regular array of CLBs, referred to as the “sandbox” by some authors [Suris et al., 2008a]. All the resources in the device are driven by a low-jitter clock-tree, with source-synchronous support and able to distribute a clock signal of up to 500 MHz across the chip. Indeed, to improve the clock distribution the Virtex-4 fabric is divided into different clock regions, or rows, as can be seen in Figure 2.10. The number of regions varies with device size, ranging from 8 in the smallest device to 24 in the largest one.

While Virtex-4 retains glitchless reconfiguration capability, its main innovation is the tile-based frame organisation of the configuration memory. In contrast to previous generations of column-based reconfigurable Virtex families, the configuration frames of Virtex-4 FPGAs are limited to the height of a clock region, spanning 16 CLBs, 32 IOBs, 8 DSP48s or 4 BRAMs. Hence, all frames in Virtex-4 have a fixed, identical length of 1312 bits (41 32-bit words). The tile-based architecture makes it possible to access a subset of resources separately in the device, thus permitting much more flexible 2-Dimensional floor-plans. Moreover, the reconfiguration speed of Virtex-4 devices is significantly faster than in previous Virtex families thanks to its improved 32-bit width and 100 MHz ICAP instance. This brings a theoretical reconfiguration throughput of 400 MB/s; i.e., a Virtex-4 configuration frame can theoretically be read and written in only 410 nanoseconds. Furthermore, it is reported that the ICAP logic can be over-clocked to achieve a higher throughput which can reach 2.2 GB/s in the newer Virtex-5 family [Shelburne et al., 2008, Claus et al., 2010, Duhem et al., 2011, Otero et al., 2011, Hansen et al., 2011, Bonamy et al., 2012].
Since the research work developed in this thesis is highly related to the technological features of the FPGA used (Virtex-4), the next subsections describe in detail the architecture of these devices. The description is aimed at providing a link among the built-in resources in the device, the hardware primitives used by Xilinx's synthesis tools and, especially, the mapping of the resources into the configuration memory. Most of the information provided was collected from several sources, including Xilinx published documentation, the hardware view of the device delivered by Xilinx FPGA_Editor design tool (see Figure 2.16) and XDL description files. The other information comes from reverse engineering experiments carried out by us.

### 2.3.1 Built-In Logic Resources and Hardware Primitives

The Virtex-4 family of FPGAs comprises three platform subfamilies: (1) LX, which is optimised for general logic applications, (2) SX, which is optimised for very high-performance signal processing applications, and (3) FX, which is optimised for embedded processing ap-
Chapter 2- Introduction to FPGAs and Dynamic Partial Reconfiguration

applications and networking. The difference between these families lies in the type and number of resources embedded in the fabric, but the general architecture of the device remains the same.

This thesis focuses on the most basic and generic types of resource which are common to all the platforms: logic, memory, clocking and interconnect. Application specific resources, such as DSP48s, are not directly addressed as they do not play a vital role in the scope of this thesis, but they are still considered due to the implications derived from their mere existence.

It is important to note the way Xilinx tools designate the resources in the device. An “X” followed by a number identifies each column of resources, counting in sequence from left to right. Likewise, a “Y” followed by a number identifies the position of each row of resources, counting from bottom to top. Hence, a resource located in the lowest left corner of the device receives X0Y0 coordinates.

**Configurable Logic Blocks (CLBs)**

CLBs are the most abundant and main logic resources in Virtex-4 FPGAs. As shown in Figure 2.13, each CLB is divided into four interconnected slices which are grouped in pairs with each pair organised as a column. SLICEM indicates the pair of slices in the left column, and SLICEL designates the pair of slices in the right column. Hence, SLICELs receive odd “X” numbers and SLICEMs receive even “X” numbers. Likewise, “Y” counts the slices starting from the bottom in sequence: 0, 1, 0, 1 (first CLB row); 2, 3, 2, 3 (second CLB row); etc.

The total number of logic resources included in one CLB are detailed in Table 2.1. As shown in Figure 2.12, each of the slices includes two 4-input Look-Up Tables (LUTs), two flip-flops, multiplexers, arithmetic gates, and fast carry-chains directly connecting to the above and below CLBs. The naming scheme used by Xilinx tools to designate these slice components refer to the LUTs and flip-flops. Hence, the lower and upper LUTs are named “F” and “G” respectively, and the lower and upper flip-flops are named “X” and “Y” respectively. Note that the peripheral logic around these components also receive the same labels.

Many of the above-mentioned slice components are the basis of some of the hardware primitives available when designing with Virtex-4 devices. For instance, the FDCE primitive
LUTs are the original and most versatile programmable components in a slice. A 4-input LUT can compute any logic function of 4 inputs by simply programming its content with the truth table of the function to implement. By aggregating several LUTs using slice multiplexers, more complicated functions, or functions with more inputs, can be computed. Some of the LUTs are featured with additional functionality. Indeed, the LUTs included in SLICEMs are “writeable” and thus, they can be used as distributed memory blocks or as shift-registers as well. For the latter specific case, dedicated lines exist that connect adjacent CLBs allowing the concatenation of various LUTs to form deeper shift-registers. LUTs used as memories can also be aggregated to achieve greater storage or to implement a dual-port memory.

As a result, four main types of hardware primitive use the same underlying LUT hardware: LUT1/2/3/4 when the LUT is used to implement a 1/2/3/4-input logic function, ROM16X1 when the LUT is used as ROM memory, RAM16X1 S/D when the LUT is used as a Single or Dual-port RAM memory, and SRL16 when the LUT is used as shift-register. It is important to note that in the last two cases the content of the LUT may change dynamically. To prevent information corruption, Virtex-4 includes the possibility of masking out LUT content when reconfiguring.

**Block RAMs**

The Virtex-4 includes dual-ported memory Block RAMs, which are organised in columns. Each BRAM offers 16Kb to store data and an extra 2Kb originally intended to store parity bits, but which can be used to store data as well.

Each port of a BRAM can be configured in any “aspect ratio” from 16K x 1, 8K x 2, to 512 x 36, and the two ports are independent even in this regard. It is the responsibility of the designer to ensure both ports do not access the same memory location at the same time. Furthermore, BRAMS are cascadable to enable a deeper and wider memory implementa-

![Table 2.1: Logic resources in one CLB](image-url)
tion, with a minimal timing penalty. BRAMs can also be used as ROM memories with the content specified in the HDL description file.

In the Virtex-4 architecture, dedicated logic in the BRAM permits the implementation of synchronous or asynchronous FIFOs. This eliminates the need for additional CLB logic for counter, comparator, or status flag generation, and uses only one block RAM resource per FIFO. The supported configurations for the FIFO are 4K x 4, 2K x 9, 1K x 18, and 512 x 36.

Finally, two vertically adjacent BRAMs in a column can be configured as a single 512 x 64 RAM with built-in Hamming Error Correction Codes (ECC). The eight protection bits are generated during each write operation, and are used during each read operation to correct any single error, or to detect (but not correct) any double error. Two status outputs indicate the three possible read results: (1) no error, (2) single error corrected, and (3) double error detected. Note that the read operation does not correct the error in the memory array, it only presents corrected data on the data port.

As a result, four main types of hardware primitive use the same underlying BRAM hardware: FIFO16 when the BRAM is used as a FIFO, RAMB32_S64_ECC when the BRAM is used as ECC protected memory, RAMB16_Sn when the BRAM is used as a single port memory with word bit-length \( n = \{1,2,4,9,18,36\} \), and RAMB16_Sn_Sm when the BRAM is used as dual-port memory with word bit-lengths in each port \( n \) and \( m \) equal to \( \{1,2,4,9,18,36\} \).
Figure 2.12: Virtex-4 SliceM
Figure 2.13: BRAM hardware structure

Programmable Routing Lines

Every resource in the Virtex-4 is connected to at least one Switch Box (SB) via a Connection Box (CB). The SBs are interconnected to each other by means of a segmented and hierarchical routing structure, called the General Routing Matrix (GRM), which consists of different types of global and local routing lines (see Figure 2.14). Global lines include **Long lines**, which span the full height and width of the device, and **Hex lines**, which connect any two SBs located at 3 or 6 positions away in the four directions (north, south, east and west). On the other hand, local lines include **Double lines**, which connect every first or second neighbour SB in the four directions, and **Direct lines**, which connect all immediate neighbour SBs. Besides, the resources are directly connected to their immediate up and down neighbours with **fast-connection lines**; e.g., carry chains and shift bits in the case of CLBs, and lines to create cascaded memories.

SBs collect most of the **Programmable Interconnections Points (PIPs)** between the lines of the GRM. According to Xilinx FPGA_Editor, each Virtex-4 SB contains up to 423 PIPs. Likewise, CBs are scaled down versions of SBs with many fewer PIPs. Neither SBs nor CBs are full crossbars as not every PIP is connected to every other PIP [Koch, 2009]. This can be seen in Figure 2.15b to Figure 2.15e, where the PIPs are located on the edges lines of the SB, and the
lines represent examples of wire connections among those PIPs. For instance, the 2 regional and 8 global input lines in the SB are connected only to the 4 output clock lines. Note that these lines make up the last part of the clock-tree. Furthermore, as it is usual that some ports of the hardware resources are permanently connected to '0' or '1' logic levels (e.g., enable signals), all of the SBs include direct access to Vdd and GND power lines by means of the so-called TIEOFF cells (see Figure 2.15a).

Physically a PIP consists of a pass transistor which is controlled by a bit in the configuration memory. By enabling the appropriate PIPs, the lines can be connected together, and eventually any signal can be routed in the device. In order to establish a connection between two lines, two PIPs must be activated in the SB: input and output [Bellato et al., 2004]. Additionally, if any of the resources connected to that SB is the source or destination of the route, two more PIPs must be activated in the CB. PIP-based “direct” interconnection permits parasite resistance and capacitance to be almost constant, resulting in predictable propagation times of the signals, which is vital for synchronous design. In [Koch, 2009], the delay of a SB hop is reported to be approximately 80 ps, having been experimentally obtained by running the delay mediator in the Xilinx FPGA_Editor.

As can be seen in Figure 2.16, the GRM is physically distributed in horizontal and vertical routing channels between each SB. Note that the routing structure of Virtex-4 is highly regular despite its resources being heterogeneous. For instance, each CLB has a dedicated
SB while each BRAM, spanning 4 CLBs in height, is connected to 4 SBs. The same applies for DSP48s (i.e., 2 DSP48s are connected to 4 SBs), and for IOBs (i.e., 2 IOBs are connected to one SB). However, fast-connection lines vary depending on the type of resource. As can be seen in Figure 2.16, DSP48s include lots of fast-connection lines to exchange intermediate results with other DSP48s, and IOBs do not include any fast-connection lines.

Clocking Resources

Figure 2.17 shows the clock-tree of a Virtex-4, including global and regional clocking resources. The global clocking resources are located in a dedicated central column in the FPGA die. 32 global clock lines are each driven by a global clock buffer (BUFGCTRL). BUFGC-
Figure 2.16: (a) Detailed view of the central part of a clock region in a Virtex-4 with the regional clocking lines splitting it in two halves. *Long* and *hex* lines are represented in dark gray color. (b) (c) (d) and (e) Detailed view of each type of Virtex-4 resource. *Long* and *hex* dark gray lines wrap each resource tile while *double* and *direct* light gray lines connect each resource to a SB via a CB.
TRLs can select between two input clock sources, either an IOB or a Digital Clock Manager (DCM). Usually the latter are preferred to eliminate the clock distribution delay, adjust its delay relative to another clock, or adjust the frequency of the clock source. A set of PIPs permit the routing of 8 out of the 32 global clock signals to the fabric, either to the regional clocking resources in each clock region or directly to the logic resources.

Figure 2.17: Simplified structure of the Virtex-4 clock-tree

Each clock region includes two regional clock buffers (BUFRs), $X_c Y_r$ and $X_c Y_r + 1$, located in the central part of the border IOB columns. BUFRs are fed with a clock signal coming from an IOB, through an I/O clock Buffer (BUFIO), or directly from a BUFGCTRL. BUFRs include the capability to divide the input clock rate by any integer number, named BUFR_DIVIDE, which can range from 1 to 8. The regional clock signal is delivered to the
resources of the clock region using two regional clock lines, A and B, which horizontally cross across the middle of the clock region. These horizontal lines have vertical branches that connect to every SB in each column. The clock source of the regional clock lines is set by means of PIPs. Indeed, it is possible to select any of 6 different input clock signals for them, 2 coming directly from the BUFRs in that clock region, \( X_c Y_{r+1} \) nd \( X_c Y_r \), and the other 4 coming from the BUFRs in the immediate adjacent up and down clock regions \( X_c Y_{r+3}, X_c Y_{r+2}, X_c Y_{r-1} \) and \( X_c Y_{r-2} \). We name as Regional Clock Multiplexer (RCMUX) the set of PIPs which select the input clock signal for the 2 regional clock lines. Note that in total each BUFR can drive up to three adjacent clock regions.

Due to the physical disposition of the clocking resources within the fabric, global resources are usually denoted as Vertical Clock (VCLK) and regional resources as Horizontal Clock (HCLK).

Summing up, every resource in the chip can be fed by any of the 8 global and 2 regional clock lines. The clock-tree of Virtex-4 is hierarchically structured as follows:

\[
\text{IOB} \rightarrow (\text{DCM}) \rightarrow \text{BUFGCTRL} \rightarrow \text{PIPs} \rightarrow (\text{BUFR}) \rightarrow \text{PIPs} \rightarrow \text{SB} \rightarrow \text{CB} \rightarrow \text{Resources}
\]

### 2.3.2 Virtex-4 Architecture Mapping to Configuration Bitstream

The configuration bitstream contains a mixture of configuration data (bits that are used to define the state of programmable resources) and configuration commands (instructions that tell the configuration logic what to do with the configuration data).

**Configuration Data**

Unfortunately only a limited description of the Virtex-4 bitstream can be found in [Xilinx Inc., 2009]. The functionality of each configuration bit composing the bitstream remains undisclosed and can only be determined by means of reverse-engineering. The vast amount of configuration information to deal with (especially related to routing) as well as the complicated relations between the hardware representation of the device offered by the Xilinx FPGA_Editor and the real underlying fabric adds an extra difficulty to this process. For this thesis, we have worked on reverse-engineering the bitstream for identifying some key configurations that need to be used by R3TOS.

---

2 In the words of Austin Lesea, senior researcher at Xilinx Inc., “Xilinx FPGA_Editor is a programmer’s invention to describe the hardware: it is a fantasy, only a convenient construction.”
Physically, the configuration memory of Virtex-4 FPGAs is distributed along the entire die, with its cells placed close to the resources being controlled [Rau et al., 2001]. Logically, the configuration memory of Virtex-4 FPGAs is organised in frames of 41 words of 32 bits. The serpentine shift register implementation of the memory means that frames in the bottom half of the device mirror images in the top half with the exception of the central 21st configuration word; i.e., configuration data is shifted in the opposite directions in each part of the device.

The addressing scheme of the frames is related to the type and location of the physical resources they configure and, hence, the configuration memory reflects the regular structure of the FPGA fabric (see Figure 2.18). The fabric is divided into two halves, top and bottom, and each of these parts is also divided into the aforementioned clock regions or rows. Each column within a row is configured by a different number of frames according to the type of resources it includes; i.e., CLB columns are configured by 22 frames, DSP48 columns require 21 frames and IOB columns need 30 frames. All these resources share the same resource block type identifier (“0”). However, BRAM configuration data consist of two separate block types. The first block contains 20 frames of routing information (block type id. “1”) and the second block includes 64 frames of BRAM content data (block type id. “2”). Most CLB, DSP48 and IOB frames enclose exclusively routing information related to the regular SBs and CBs and hence, they are identical in structure to most BRAM interconnection frames. The remaining frames comprise configuration information related to the non-regular routing lines and, hence, they are specific for each type of resource.

As shown in Table 2.2, every frame is addressed by a 23-bit address which includes five values referred to the resources it configures: (1) block resource type, (2) top / bottom half, (3) row, (4) major column address, which identifies the column within the row and increments from left to right, and (5) minor intra-column address, which identifies the specific frame within the column.

Figure 2.19 depicts an exemplary frame addressing scheme. In this figure the correlation between the configuration data and the underlying physical structure of the fabric can be easily seen. For instance, the global clocking resources located in the centre of the fabric are mapped to a dedicated VCLK column in the middle of the bitstream. It must be noted that two pad frames are added at the end of each row.
This correlation is more evident when focussing on the internal organisation of the configuration frame. Note in Figure 2.20 that the HCLK resources located in the middle of the clock regions are mapped into the middle word of the frames (21st word). Moreover, each CLB row within a column is mapped to an equivalent frame segment of 80 bits (i.e., 2 and a half words), up and down of the HCLK word. The first CLB row is mapped within [0:79] frame bits, the next CLB row is mapped within [80:159] bits, etc. Due to the mirroring between top and bottom half frames, these segments are mapped to CLB rows in opposite order in both parts of the device. Segments in the frames with MINOR>18 contain the logic configuration of the CLBs, while segments in the frames with MINOR<19 include the routing configuration.
The same applies to BRAMs, DSP48s and IOBs, but as these resources are not associated with the SBs in one-to-one fashion, the logic and routing configurations are mapped to frame segments of different length. Specifically, logic segments of BRAMs are 320 bits length, of DSP48s are 160 bits length, and of IOBs are 40 bits length, while routing segments are for all of them, as for CLBs, 80-bit length.

Figure 2.21 shows the way LUTs and flip-flops are mapped within the CLB configuration frames. Frames with MINOR=19 configure SLICEL-LUTs and frames with MINOR=21 configure SLICEM-LUTs. Both frames have the same internal structure. It must be noted that LUT data is stored in the true sense, but the bit order is swapped; e.g., bit position 0 in a
configuration frame corresponds to the bit which is accessed when the address of the FLUT at \(X_c, Y_r\) is set to “1111”, while bit position 15 in the frame corresponds to the bit accessed when the address of that FLUT is set to “0000”. The configuration mode of SLICEM-LUTs (i.e., RAM, ROM, SRL16 or logic LUT) is defined in the frame with MINOR=20.

The frame with MINOR=20 also allocates the 128 flip-flops of a CLB column, including both SLICEL and SLICEM flip-flops. In contrast to LUTs, which are originally designed to store configured logic functionality, flip-flops are intended to store user information. This information is indeed kept in a different plane from the configuration; i.e., the mapping of the flip-flops in the configuration memory can be seen as a “shadow copy” of their actual value. Furthermore, it is important to note that flip-flop data is stored in its inverted sense in the configuration memory. Tedious operations are needed to move data between the two planes in order to be accessed from the configuration port, which has only access to the standard configuration plane. These operations include GCAPTURE, to pull data down to the configuration plane (i.e., the INIT value is overwritten with the actual captured value), and GRESTORE, to push data up to the user information plane (i.e., flip-flop data is overwritten with the value in the INIT bits). By default GCAPTURE and GRESTORE operate over all the flip-flops in the device, but it is possible to restrict the operation to only a subset of flip-flops by setting-up a special mask register, which is also stored in the configuration memory. The mask bit for (all of) the flip-flops contained in an FPGA column is located in the 13th position within the frame whose address includes the top/bottom, row and column values corresponding to the location of the flip-flops, block type id. “3” and MINOR=0. The GCAPTURE and GRESTORE operations only apply to the subset of flip-flops whose associated mask bit is equal to '0'. Additionally, when a reset is asserted on a flip-flop, this loads its default value, which is mapped to SRMODE bit; i.e., when SRMODE='0' the flip-flop is initialised to logic level '1' and vice versa.

Figure 2.22 shows in more detail the way BRAM content is mapped within the configuration frames. Basically, 4 BRAMs (72 Kb) are mapped to 64 content frames while 128 LUTs (2 Kb) are mapped to two CLB frames. Each of the 64 BRAM content frames stores 256 data and 32 parity bits of each of the 4 memories mapped to that frame; i.e., the frame with MINOR=0 stores the first 256 data bits of each memory (INIT_00 in the VHDL code), frame with MINOR=1 stores 257 to 511 data bits (INIT_01 in the VHDL code), etc. This means that
Figure 2.21: LUT and flip-flop mapping in top half configuration frames. Note that \(r\) and \(c\) represent the row and column of the slice.
Figure 2.22: BRAM content data mapping in top half configuration frames (All minor addresses).
Note that \( r \) and \( c \) represent the row and column of the BRAM.
when the BRAM is configured as 4K x 4, each frame stores up to 64 positions of each memory and when the BRAM is configured as 512 x 64 with ECC protection, each frame stores only 4 memory positions. Moreover, it is important to note that, in contrast to LUT bits, BRAM content bits are not in the correct order within the configuration frame.

As with flip-flops, BRAM content data is kept in the user information plane. However, due to the relatively large size of BRAM memories, there is no “shadow copy” of their content into the configuration plane and thus, GCAPTURE and GRESTORE operations do not apply to BRAMs. Instead, the content of BRAMs can be directly accessed in the user information plane through the configuration port. However, as one of the ports of the BRAM needs to be used to implement the interface between both planes, data corruption may occur when accessing a BRAM simultaneously from the design and from the configuration layer; i.e., when the two ports are used in the design (BRAM configured as dual-ported) one of the ports is inevitably shared between the design and the configuration interface, provoking undetermined functioning. Writing from the configuration port to the user information plane is protected by the SAVEDATA bits, which must be set to '0' for writing to be permitted. Indeed, these bits enable individual access to each BRAM.

Finally, it is worth concentrating only on a subset of the vast routing resources of the FPGA, the clock-tree, which is indeed one of the most important innovations in the Virtex-4 architecture, playing a vital role in synchronous design. As previously introduced, VCLK is mapped to dedicated frames while HCLK resources are mapped to the 21st word of block type id. “0” frames. Specifically, the two BUFRs in each clock region are mapped to bit positions [652:655] of the border IOB frames with MINOR=14 and MINOR=22, respectively, and the PIPs to select the clock source for the two regional clock lines (i.e., RCMUXes) are mapped to the same bit positions of the IOB frames with MINOR=23 and MINOR=25, respectively. Table 2.3 shows the configuration values to select the clock rate at the output of the BUFR (BUFR_DIVIDE parameter value) and Table 2.4 shows the values to select a specific clock source for the regional clock lines (i.e., RCMUX configuration). BUFRs can be individually switched off and on by writing ‘0’ or ‘1’ in bit positions 652 (for $X_c Y_i$) and 653 (for $X_c Y_{i+1}$) within the border IOB frame with MINOR=23. Likewise, both BUFRs can be switched off and on at the same time by writing ‘0’ or ‘1’ in bit position 655 within the latter frame.
Table 2.3: BUFR frequency selection

<table>
<thead>
<tr>
<th><strong>BUFR_DIVIDE</strong></th>
<th><strong>Value</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bypass</td>
<td>0x0</td>
</tr>
<tr>
<td>$f_{input}/f_{output} = 1$</td>
<td>0x8</td>
</tr>
<tr>
<td>$f_{input}/f_{output} = 2$</td>
<td>0x9</td>
</tr>
<tr>
<td>$f_{input}/f_{output} = 3$</td>
<td>0xA</td>
</tr>
<tr>
<td>$f_{input}/f_{output} = 4$</td>
<td>0xB</td>
</tr>
<tr>
<td>$f_{input}/f_{output} = 5$</td>
<td>0xC</td>
</tr>
<tr>
<td>$f_{input}/f_{output} = 6$</td>
<td>0xD</td>
</tr>
<tr>
<td>$f_{input}/f_{output} = 7$</td>
<td>0xE</td>
</tr>
<tr>
<td>$f_{input}/f_{output} = 8$</td>
<td>0xF</td>
</tr>
</tbody>
</table>

Table 2.4: Clock source selection for the regional clock lines (RCMUX)

<table>
<thead>
<tr>
<th><strong>Clock source</strong></th>
<th><strong>Value</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFR $X_rY_{r+3}$</td>
<td>0x8</td>
</tr>
<tr>
<td>BUFR $X_rY_{r+2}$</td>
<td>0x9</td>
</tr>
<tr>
<td>BUFR $X_rY_{r+1}$</td>
<td>0xA</td>
</tr>
<tr>
<td>BUFR $X_rY_r$</td>
<td>0xB</td>
</tr>
<tr>
<td>BUFR $X_rY_{r-1}$</td>
<td>0xC</td>
</tr>
<tr>
<td>BUFR $X_rY_{r-2}$</td>
<td>0xD</td>
</tr>
</tbody>
</table>

Table 2.5 sums up the PIPs involved in the routing of the regional clock lines to the resources. Refer to Figure 2.17 for a simplified scheme of the Virtex-4 clock-tree. The PIPs to enable each of the vertical branches of the regional clock lines that deliver the clock signal to all the SBs in a column, are mapped to positions 655 (Line A) and 654 (Line B) within each of the CLB, DSP48 and BRAM interconnection frames with MINOR=18. The PIPs that control the routing of the clock signal inside the SBs and CBs are mapped within the frames with MINOR=14 and MINOR=15. We have found out that up to 12 PIPs are necessary to route the clock signal inside each SB and CB. Note that this contrasts with the expected number of involved PIPs according to the representation shown in Xilinx FPGA_Editor (see Figure 2.15): 2 inputs + 4 outputs in the SB, and another 4 inputs + 4 outputs in the CB, that is, in total 14 PIPs. This is due to the existing deviation between the representation shown by Xilinx tool and the real implementation in the FPGA die.

**Configuration Logic and Commands**

It is interesting to the scope of this thesis to analyse the configuration commands and the format of the configuration bitstream as they directly refer to the configuration logic in the device.

The configuration logic can be accessed through different interfaces (e.g., internal ICAP and external JTAG, SelectMAP or serial port). Aiming at building a true SoC, this thesis uses the ICAP interface. However, note that the ICAP can only be used after the initial configuration of the system has been uploaded to the FPGA, from a non-volatile memory using the
SelectMAP interface or, typically, from a PC using the JTAG interface. To select the active configuration interface, three mode pins (M0..M2) are provided.

The Virtex-4 ICAP has a chip-select signal (CS), a busy signal (BUSY), a read-write control signal (RD), a clock (CLK), a write data bus (DIN), and a read data bus (DOUT). Data buses can be configured to 8 or 32 bits width, and one data word is transferred on each clock edge. It is important to note that there are two ICAP sites in Virtex-4 devices: TOP and BOTTOM, both located in the central column on the die. However, the two interfaces can never be used at the same time because they share the same underlying configuration logic. The TOP site is active after configuration by default, but then it is possible to switch to the BOTTOM site.

On its part, Virtex-4 configuration logic consists of a packet processor, a set of 32-bit configuration registers, and global signals that are controlled by the registers (see Table 2.6). The packet processor controls the flow of data from the configuration interface to the appropriate register and vice versa. Indeed, all configuration commands are executed by reading or writing to the configuration registers. The most important commands are shown in Table 2.7.

<table>
<thead>
<tr>
<th>MINOR</th>
<th>Bit position</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>655</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>40+4n+80m</td>
<td>0 ≤ n ≤ 3</td>
<td>0 ≤ m ≤ 7</td>
</tr>
<tr>
<td>72+4n+80m</td>
<td>0 ≤ n ≤ 3</td>
<td>9 ≤ m ≤ 15</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>45+8n+80m</td>
<td>0 ≤ n ≤ 1</td>
<td>0 ≤ m ≤ 7</td>
</tr>
<tr>
<td>46+8n+80m</td>
<td>0 ≤ n ≤ 1</td>
<td>9 ≤ m ≤ 15</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>654</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>43+4n+80m</td>
<td>0 ≤ n ≤ 3</td>
<td>0 ≤ m ≤ 7</td>
</tr>
<tr>
<td>75+4n+80m</td>
<td>0 ≤ n ≤ 3</td>
<td>9 ≤ m ≤ 15</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>40+8n+80m</td>
<td>0 ≤ n ≤ 1</td>
<td>0 ≤ m ≤ 7</td>
</tr>
<tr>
<td>41+8n+80m</td>
<td>0 ≤ n ≤ 1</td>
<td>9 ≤ m ≤ 15</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>72+8n+80m</td>
<td>0 ≤ n ≤ 1</td>
<td>9 ≤ m ≤ 15</td>
</tr>
<tr>
<td>73+8n+80m</td>
<td>0 ≤ n ≤ 1</td>
<td>9 ≤ m ≤ 15</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: PIPs to route the regional clock signals to the resources in top half configuration frames

The Virtex-4 ICAP has a chip-select signal (CS), a busy signal (BUSY), a read-write control signal (RD), a clock (CLK), a write data bus (DIN), and a read data bus (DOUT). Data buses can be configured to 8 or 32 bits width, and one data word is transferred on each clock edge. It is important to note that there are two ICAP sites in Virtex-4 devices: TOP and BOTTOM, both located in the central column on the die. However, the two interfaces can never be used at the same time because they share the same underlying configuration logic. The TOP site is active after configuration by default, but then it is possible to switch to the BOTTOM site.

On its part, Virtex-4 configuration logic consists of a packet processor, a set of 32-bit configuration registers, and global signals that are controlled by the registers (see Table 2.6). The packet processor controls the flow of data from the configuration interface to the appropriate register and vice versa. Indeed, all configuration commands are executed by reading or writing to the configuration registers. The most important commands are shown in Table 2.7.
Chapter 2- Introduction to FPGAs and Dynamic Partial Reconfiguration

<table>
<thead>
<tr>
<th>Name</th>
<th>Read/Write</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC</td>
<td>Read/Write</td>
<td>00000</td>
<td>CRC Register</td>
</tr>
<tr>
<td>FAR</td>
<td>Read/Write</td>
<td>00001</td>
<td>Frame Address Register</td>
</tr>
<tr>
<td>FDRI</td>
<td>Write</td>
<td>00010</td>
<td>Frame Data Register Input</td>
</tr>
<tr>
<td>FDRO</td>
<td>Read</td>
<td>00011</td>
<td>Frame Data Register Output</td>
</tr>
<tr>
<td>CMD</td>
<td>Read/Write</td>
<td>00100</td>
<td>Command Register</td>
</tr>
<tr>
<td>CTL</td>
<td>Read/Write</td>
<td>00101</td>
<td>Control Register</td>
</tr>
<tr>
<td>MASK</td>
<td>Read/Write</td>
<td>00110</td>
<td>Mask register for CTL</td>
</tr>
<tr>
<td>STAT</td>
<td>Read</td>
<td>00111</td>
<td>Status Register</td>
</tr>
<tr>
<td>COR</td>
<td>Read/Write</td>
<td>01001</td>
<td>Configuration Option Register</td>
</tr>
<tr>
<td>MFWR</td>
<td>Write</td>
<td>01010</td>
<td>Multiple Frame Write</td>
</tr>
<tr>
<td>IDCODE</td>
<td>Read/Write</td>
<td>01100</td>
<td>Device ID Register</td>
</tr>
</tbody>
</table>

Table 2.6: Most important configuration registers

<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0000</td>
<td>Null command</td>
</tr>
<tr>
<td>WCFG</td>
<td>0001</td>
<td>Write Configuration Data</td>
</tr>
<tr>
<td>MFWR</td>
<td>0010</td>
<td>Multiple Frame Write</td>
</tr>
<tr>
<td>RCFG</td>
<td>0100</td>
<td>Read Configuration Data</td>
</tr>
<tr>
<td>RCRC</td>
<td>0111</td>
<td>Reset CRC register</td>
</tr>
<tr>
<td>GRESTORE</td>
<td>1010</td>
<td>Push data up to the user information plane</td>
</tr>
<tr>
<td>GCAPTURE</td>
<td>1100</td>
<td>Pull data down to the configuration plane</td>
</tr>
<tr>
<td>DESYNC</td>
<td>1101</td>
<td>Desynchronises the device</td>
</tr>
</tbody>
</table>

Table 2.7: Most important configuration command codes

The bitstream is organised in packets. There are two different types of packets: Type1 and Type2, both consisting of a 32-bit header followed by a number of 32-bit data words (see tables 2.8 and 2.9). The operation to perform (e.g., NOP: “00”, Read: “01”, Write: “10”) as well as the amount of data words is specified in the packet header. Type1 packets are used to write small blocks of data (e.g., a few configuration frames or a single register). On the other hand, Type2 packets are used to write long blocks of configuration data (e.g., all the configuration frames in a device) and must be always preceded by a Type1 packet which specifies the register address to write the data in.

Table 2.8: Packet Type1 header format

- The bitstream is organised in packets. There are two different types of packets: Type1 and Type2, both consisting of a 32-bit header followed by a number of 32-bit data words (see tables 2.8 and 2.9). The operation to perform (e.g., NOP: “00”, Read: “01”, Write: “10”) as well as the amount of data words is specified in the packet header. Type1 packets are used to write small blocks of data (e.g., a few configuration frames or a single register). On the other hand, Type2 packets are used to write long blocks of configuration data (e.g., all the configuration frames in a device) and must be always preceded by a Type1 packet which specifies the register address to write the data in.
Chapter 2- Introduction to FPGAs and Dynamic Partial Reconfiguration

<table>
<thead>
<tr>
<th>Header type</th>
<th>OpCode</th>
<th>Word Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:29]</td>
<td>[28:27]</td>
<td>[26:0]</td>
</tr>
<tr>
<td>010</td>
<td>RR</td>
<td>xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx</td>
</tr>
</tbody>
</table>

Table 2.9: Packet Type2 header format

The bitstream is protected with a standard CRC32C checksum algorithm. This algorithm operates on the 37-bit values which are formed by attaching the 5-bit register address specified in the header of write packets to each of the 32-bit words included in the packet body.

After initialisation, the packet processor ignores all data presented on the configuration interface until it receives a synchronisation sequence. This sequence includes a dummy word (0xFFFFFFFF) and a synchronisation word (0xAA995566). After synchronisation, the packet processor waits for a valid packet header to begin the configuration process.

The first configuration operations in the bitstream typically include resetting the CRC computation, setting up the reconfiguration options and control parameters (e.g., mask or not mask out the LUTs when used as SRL16 and distributed RAM) and checking the target device identifier. Indeed, the device ID is checked prior to writing any configuration data to FPGA’s configuration memory.

Configuration information is transferred immediately after having enabled the circuit to write in the configuration memory with the WCFG command. The transfer is done frame by frame and word by word until reaching the number of words specified in the packet header. The starting address for the first frame is loaded to the FAR register and then, configuration words are sequentially written to the FDRI register. Note that the address in the FAR register is automatically incremented by the own configuration logic after writing each frame. In addition to the useful data, a pad frame must be written to FDRI in order to shift in the last frame of a continuous range of frames.

After transmitting the whole configuration data to the configuration interface, all of the flip-flops in the FPGA are initialised with the INIT value by issuing a GRESTORE command. Alternatively, this operation, as well as GCAPTURE, can be triggered by directly toggling the corresponding internal signals in the configuration logic. The access to these internal

\[3\text{CRC32C polynomial: } x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{20} + x^{19} + x^{18} + x^{16} + x^{14} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^6 + 1.\]
signals is provided by two interfacing primitives to be instantiated in the design: CAPTURE (for GCAPTURE) and STARTUP (for GRESTORE).

After the GRESTORE command, the CRC value associated to the bitstream is written to the CRC register. If there is any discrepancy between the expected CRC value and the CRC value computed by the device, the configuration is aborted. This is useful to prevent loading a wrong configuration, but if not needed, CRC checking can be disabled in the COR register. If CRC is disabled the expected CRC value by the device is predetermined and equal to 0x0000DEFC. Finally, the last configuration operation in the bitstream is to desynchronise the device toggling a DESYNC command in the CMD register. This sets the packet processor in a state which waits for the synchronisation sequence and discards all other data.

In order to speed-up the reconfiguration process, configuration data can be compressed in the bitstream. Compression consists in using MFWR commands to write identical frames to multiple configuration memory address locations and hence, the achievable improvement depends on the number of identical frames in the partial bitstream.

Besides uploading new configuration data to the device, already configured data can be downloaded from it. Note that the circuit to read from the configuration memory must be first activated with the RCFG command. Analogously as with the writing process, the address of the starting frame to be read-back is loaded in the FAR register. After having delivered a pad frame and a pad word, the configuration logic makes configuration data words available in the FDRO register one by one until the number of words to be read specified in the packet header is reached. Hence, the number of words to be read must account for the additional pad information, in total 42 extra words. As shown in Figure 2.23, every ICAP transfer starts with a writing operation, even when performing a read-back operation. Indeed, note that the starting FAR address and the number of words to be read are specified in this writing operation. Then, the ICAP is changed to read mode by asserting the RD signal while CS is kept high. Valid read-back data are available at DOUT port when BUSY='0'.

Finally, we would like to point out that the pad information and configuration commands introduce an overhead that reduces the exploitable reconfiguration bandwidth. This overhead is more significant when reading or writing individual frames and it tends to be neglectful as the (consecutive) configuration data blocks are larger.
2.3.3 Reliability Concerns

As with any other semiconductor device, FPGAs are susceptible to both transient faults (e.g., radiation induced soft-errors) and permanent faults (e.g., aging provoked irreversible damage in the device silicon substrate). Indeed, the reliability of semiconductor devices is represented with a curve with the shape of a "bathtub". This curve, shown in Figure 2.24, can be divided into three main regions: (1) initial failures, which occur shortly after the device is manufactured due to latent defects, (2) random failures, which spontaneously occur over a long period of time (the lifetime of the device), and (3) wear-out failures, which progressively degrade the device due to wear and fatigue. With the current trend for smaller feature sizes and higher power density, the aforementioned bathtub curve is shifting in such a way that the effective lifetime of the devices is shortening and the random failure rate is increasing [Chandra, 2009]. While a lifetime of more than 100 years is assigned to 180 nm process technology, a lifetime shorter than 15 years is assigned to 65 nm [White and Chen, 2008].
Soft-Errors

Soft-errors are currently the predominant cause of random failures in FPGAs. They refer to changes in the logical state of one or various transistors in the device mainly provoked by radiation and, in less measure, by random noise or signal integrity problems, such as crosstalk. Note that in most of the cases soft-errors are correctable by turning the affected transistors back to their original state. The most famous radiation induced soft-errors are Single Event Upsets (SEUs), which are provoked due to the nondestructive electron-hole pairs caused by an energetic particle that strikes the chip (see Figure 2.25) [Dyer, 2001, Dodd and Massengill, 2003, Karnik et al., 2004].

Figure 2.25: Single Event Upset (SEU)

Soft-errors may provoke harmless glitches in combinatorial logic, but become a real hazard when they are captured in flip-flops or memories, or in case they directly affect any storage component, including the configuration memory itself. Indeed, the configuration cells used in Xilinx FPGAs are similar to the 6-transistor storage cells used in SRAMs, which has been proven to be sensitive to SEUs [Ohlsson et al., 1998] (see Figure 2.26). As a consequence, not only user information stored in flip-flops, BRAMs or distributed memory may be corrupted, but also the functionality of the implemented circuit may be changed; i.e., change the routing or the logic configuration of the hardware resources (see Figure 2.27) [Graham et al., 2003, Bellato et al., 2004, Kastensmidt et al., 2006]. SEUPI is a tool developed by Xilinx Inc. to identify all the configuration bits which are potentially sensitive to SEUs in a particular design (i.e., only the “used” bits in that design are considered), thus preventing expensive in-field tests [Sundararajan et al., 2003]. To do so, SEUPI includes the knowledge of the mapping between the configuration bits and the underlying FPGA architecture.
The real results measured in the Rosetta experiment show 263 FIT/Mb $^4$ in the Virtex-4 configuration memory and 484 FIT/Mb in BRAMs due to radiation provoked soft-errors [Xilinx Inc., 2012a].

To reduce the effect of soft-errors ECCs can be used. As previously presented, Virtex-4 BRAMs are fitted with built-in ECC logic, and the configuration information is also protected with ECC. Namely, each configuration frame includes a 12-bit SEC-DED Hamming code in the 21st word from 640 to 651 bit positions (see Figure 2.20). These codes are automatically produced by Xilinx bitstream generation tools, assuming the configuration data will remain the same during system functioning. Note that user information (e.g., LUTs used as SRL16 or as distributed RAM) is masked out with zeros during the ECC computation process. Indeed, since the BRAM content frames only store user information they do not include any ECC codes.

It is important to the scope of this thesis to know the way ECC bits are computed in order to enable the modification of the configuration information at runtime without losing the protection; i.e., ECC bits must be updated when the configuration is changed. To compute the ECC on a configuration frame, its data bits are expressed in 32-bit words indexed

---

$^4$One FIT is equal to one failure per $10^9$ hours of system operation.
by an integer number $N$ [Warren et al., 2008]. Hence, the range of data bits is $D(32 \cdot N)$ to $D(32 \cdot N + m)$, where $N$ is the word index and $m$ is the bit index within each word. $N$ ranges from 22 to 31 and from 33 to 63. $N=32$ is skipped because of ECC’s algorithm issues as it is a power of two. $m$ ranges from 0 to 31 for all $N$s, except for $N = 43$ when it ranges from 0 to 19. The remaining 12 bits in this word are used to store the ECC bits as $N = 43$ corresponds with the 21st word in the frame. Therefore, data is organised in segments as follows: $D(704):D(1023), D(1056):D(1395), ECC(0):ECC(11), D(1408):D(2047)$. Table 2.10 shows the exemplary embodiment of data bit position matrix for a generic 32-bit data word $D(32 \cdot N + m)$. In this context each bit position within a frame can be coded using 11 bits, $N_5,...N_0,m_4,...m_0$. These values are used to determine whether a given data bit takes part or not in the computation of ECC(1):ECC(11); i.e., ECC(1):ECC(5) exclusively depend on the position of the data bit within the data word ($m$), while ECC(6):ECC(11) exclusively depend on the position of the data word within the frame ($N$). Note that $m$ is substituted with the actual binary values in Table 2.10. The data bits whose position is coded with a ‘1’ in the corresponding $N$ or $m$ bits are XORed to calculate the ECCs, e.g., only the data bits in odd positions take part in the computation of ECC(1). On its part, ECC(0) is the parity bit of the whole frame. In this situation, the calculation of ECC bits in a frame is an iterative process, which starts with the initialisation of the ECC bits to ‘0’ value. For each word, selected subsets of its 32 bits are XORed together, as directed by the matrix entries shown in Table 2.10. The resulting ECC bits are XORed with the previously calculated ECC bits for precedent data words, giving rise to new ECC bits. This process is repeated until the last word in the frame is processed. Finally, the ECC values are XORed together and with ECC(0), yielding the final value for ECC(0).

Xilinx Virtex-4 FPGAs include a built-in logic coupled to their configuration memory which automatically checks the ECCs each time a frame is read-back (see Figure 2.28). During read-back, this logic calculates a syndrome value using all the bits in the frame, including the ECC bits. Note that user information must be masked during the read-back process. This is done by writing a ‘1’ to the 8th bit of the MASK register followed by writing a ‘0’ to the 8th bit of the CTL register (GLUTMASK_B bit). At the end of each read-back, the SYNDROME_VALID signal is asserted for one cycle. If the bits have not changed from the original values, then SYNDROME are all ‘0’s. Otherwise, the ERROR output of the Frame_ECC
<table>
<thead>
<tr>
<th>ECC(0)</th>
<th>ECC(1)</th>
<th>ECC(2)</th>
<th>ECC(3)</th>
<th>ECC(4)</th>
<th>ECC(5)</th>
<th>ECC(6)</th>
<th>ECC(7)</th>
<th>ECC(8)</th>
<th>ECC(9)</th>
<th>ECC(10)</th>
<th>ECC(11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.10: Frame_ECC Polynomial
logic is asserted. If a single bit has changed, including any of the ECC bits, then SYN-
DROME(11) is ‘1’ and the position of the corrupted bit is coded by SYNDROME(10..0). How-
ever, it is important to note that the Frame_ECC logic is not correctly coupled to Virtex-4
configuration memory, as it operates one clock cycle ahead. This means that when reading-
back a frame, the calculated syndrome is not the expected as it also includes the first word
of the next frame.

To decode the syndrome value to an index in the range 0 to 1311, a 704 value must be
subtracted if SYNDROME(10..0) is less than 1024 decimal and, if it is not, a 736 value must
be subtracted. Note that a single bit error can be corrected by simply updating the corrupted
bit with its inverted value. If two bits have changed, then SYNDROME(11) is ‘0’ and SYN-
DROME(10..0) is non-zero and meaningless. In this case, a “golden copy” of the affected
frame must be written-back. The technique that consists in rewriting the configuration
memory to correct soft-errors is known as Scrubbing [Charmichael, C. and Tseng, C. W.,
2009, Heiner et al., 2009]. Blind scrubbing is a simpler approach which consists in peri-
odically rewriting the complete golden configuration of the FPGA without carrying out any
previous diagnosis; i.e., Frame_ECC logic is not used. Note that the scrubbing rate should
be determined based on the expected soft-error rate. As newer Xilinx FPGAs include a ded-
icated logic to automatically and continuously check the integrity of the configuration data
without requiring any intervention from the user design, blind scrubbing is triggered only
when a soft-error is detected.

Xilinx Inc. released a controller macro to implement SEU detection and correction using
Frame_ECC and ICAP [Chapman, 2010], while NASA's Goddard Space Flight Center (GSFC)
presented an external controller to implement blind scrubbing using a golden configuration bitstream and SelectMAP [Berg et al., 2008]. When the external controller is implemented in a hardened device (e.g., an ASIC), the latter scheme is more reliable than when using soft-error prone Xilinx controller internally implemented in the SRAM-based FPGA. Indeed, in the radiation tests carried out in [Berg et al., 2008] additional upsets were discovered presumably provoked by the internal scrubber itself. To make the internal controller highly reliable, a set of fault-tolerance techniques are presented in [Heiner et al., 2008]. Finally, it is important to note that the Xilinx SEU controller macro is the only known solution that solves the existing Frame ECC malfunction in Virtex-4 devices.

Another approach in light of a higher reliability consists on using redundant components that mask the faults, guaranteeing the continuous working of the system (e.g., Triple Modular Redundancy - TMR). Unfortunately, redundant components consume hardware resources that provide no performance benefit and lead to higher power consumption. In [Sterpone and Violante, 2005, Golshan and Bozorgzadeh, 2009] the redundant components are floor-planned on separate locations in the chip’s substrate in order to minimise correlated faults. Xilinx has recently developed the XTMR Tool to automate the hardening of a design by triplicating the input/outputs, the throughput logic, and by inserting feedback logic for registered data correction [Xilinx Inc., 2007b]. Another automatic tool to create TMR designs is described in [Pratt et al., 2006]. This tool triples the logic according to the reliability requirements of each part of the system.

By combining scrubbing, ECC-protected BRAMs and TMR the failure rate of a circuit can be reduced to single-digit FITs [Lesea and Alfke, 2008, Bridgford et al., 2008]. However, FPGAs include circuitry that cannot be scrubbed, namely the power on reset, DCMs and the configuration logic itself. A single soft-error affecting these parts stops the system operation until the chip is reset. This is known as Single Event Functional Interrupt (SEFI) and it is reported a very low occurrence rate in the order of $10^{-7}$ SEFI/day [Charmichael, C. and Tseng, C. W., 2009]. SEFIs can be detected by means of simple diagnostic tests, such as writing and reading a predefined value to the FAR register, or monitoring the value of some specific pins of the FPGA. In critical applications SEFIs can be mitigated by using redundant FPGA devices [Bridgford et al., 2008].
As a final note in this section we point out that FPGA vendors do manufacture some reconfigurable technologies with special hardness to soft-errors (e.g., antifuse and flash FPGAs). However as these are very expensive and show integration, speed, and consumption limitations [Battezzati et al., 2011], soft-error sensitive SRAM-based FPGA is currently the mainstream FPGA technology and the main focus of this research.

**Permanent Hardware Damage**

The ever smaller feature sizes of current FPGAs make them more vulnerable to suffering from semiconductor degradation mechanisms such as Electromigration (EM) [Christou, 1994, Lienig, 2006], Hot Carrier Injection (HCI) [Hu et al., 1985], Time Dependent Dielectric Breakdown (TDDB) [Li et al., 2008] and Negative Bias Temperature Instability (NBTI) [Chakravarthi et al., 2004, Kumar et al., 2006]. These “wear-out” mechanisms provoke irreversible damage to the silicon substrate of the chip (see Figure 2.29) and get gradually worse as it ages. Note that these mechanisms can also lead to early-life failures when they lay stress on the defects produced during the manufacturing process of the devices. Permanent damage can be an important hazard in some specific applications, such as safety or security critical applications. For instance, in [Ho et al., 2011] the authors show that an adversary with knowledge of a fault in a microprocessor can launch attacks which can obtain critical secrets such as a private key in as short as 30 seconds. In any case, wear-out failures are not considered during the lifetime of a device in most of applications. However, they are predicted to become a major problem in the future [ITRS, 2011] and, even today there are applications where device lifetime is a subject of concern (e.g., deep space exploration missions [Iturbe et al., 2011b]). In [Welling, 2010], the authors prove that the energy deposited onto a spacecraft due to high energy particles and electromagnetic radiation in space has the ability to break down its electronics, resulting in operational anomalies and accelerated wear-out.

The rate of degradation experienced by an FPGA device has been proven to depend on diverse factors, such as supply voltage, operating temperature and switching activity [Stott et al., 2010, Stott, 2011, Mehta and DeHon, 2011, Amouri and Tahoori, 2012]. The latest reliability report made available by Xilinx Inc. in 2012 confirms the natural rule that FPGA chips are more prone to suffering a hardware fault as they use smaller features: the officially
reported FIT rate for 90 nm process technology (used by Xilinx-4 FPGAs) is only 5 FIT, but it is 13 FIT for 65 nm (Virtex-5), and raises up to 24 FIT for latest 28 nm (Virtex-7) [Xilinx Inc., 2012a]. Furthermore, the study carried out in [Srinivasan et al., 2006] demonstrates that an FPGA device usually starts to fail rather early under continuous operating conditions, projecting a significant FIT value due to permanent damage. Unfortunately, it is not possible to predict with precision the way at which a given device is degraded nor the consequences this degradation will involve to a specific application. Hardware faults can be detected, and the affected resources identified only at runtime.

Specific diagnostic hardware can be included in a chip to detect damage, i.e., Built-In-Self-Tests (BISTs) [Dutt et al., 2008]. In [Constantinides et al., 2007], the authors report a damage detection rate as high as 99.22% using a hardware that gives access and control to a microprocessor's internal state. Since the number of components that perform the key operations on a microprocessor are limited, this diagnostic hardware can be kept very simple (e.g., only 5.8% area increase in [Constantinides et al., 2007]). However, extending this idea to the vast amount of logic and routing resources of an FPGA is not trivial and results in large area overheads [Smith et al., 2006, Amouri and Tahoori, 2011].

Fortunately, there are other ways to detect permanent damage in an FPGA. As the configuration cells of Xilinx FPGAs are distributed along the chip, it is likely that any damage provoked in the fabric affects at least one of these cells, resulting thus in some configuration bits stuck-at '0' or '1'. These bits can be detected using the frame ECC codes or comparing the read-back configuration data with the correct data. Based on the location of the stuck-at bits within the configuration memory (i.e., frame address and bit offset within the frame) it is possible to identify where the damage is located in the chip. Then, the system can be appropriately readjusted by using DPR not to include the damaged resources. In [Sundararajan
and Guccione, 2001, Paulsson et al., 2006, Montminy et al., 2007], the pieces of circuitry that use damaged resources in their original locations are reallocated to other positions where all of the resources that need to be used are operational. On the other hand, in [Iturbe et al., 2009, Wirthlin et al., 2013], a battery of diverse circuit implementations that use different sets of resources are created at design time and used as needed as the FPGA gets damaged.

In [Lohn et al., 2003, Garvie and Thompson, 2004] a different method to cope with the permanent damage is used. This method does not require any previous diagnosis nor any knowledge of bitstream format as it relies on an Evolutionary Algorithm (EA) to blindly explore the bitstream space in order to find an alternative configuration for which the partially damaged FPGA implements the same functionality. However, the convergence of the EA is limited by the complexity of the system where it is used [Lohn et al., 2003, Greenwood and Tyrrell, 2006].

Biology has been the inspiration of other research efforts. In order to build electronic systems with the fault-tolerant and surviving properties of living beings, researchers have tried to mimic in electronics some of the biological characteristics and processes that occur at cellular and subcellular levels. For instance, an “electronic immune system” to detect the “pathogens” (i.e., faults) that cause “disease” (i.e., malfunctions) was proposed in [Bradley and Tyrrell, 2000, Avizienis, 2002], an “electronic tissue” with self-healing capabilities (i.e., cellular replication, specialisation and substitution) was developed in [Tyrrell and Sun, 2006, Tempesti et al., 2007], and the “electronic DNA” (eDNA) concept as a mechanism to create self-organising systems was demonstrated in [Reibel Boesen, 2011, Samie et al., 2011]. It is important to note that the key representative features of biological organisms (e.g., redundancy, flexibility, adaptability and autonomy) are inherent or can be developed using massively parallel and DPR-enabled FPGAs and thus, these devices are used in most of biologically inspired works [Iturbe et al., 2010c].

2.4 Dynamic Partial Reconfiguration

While previous sections have described the architecture of modern partially reconfigurable Xilinx FPGAs, this section is aimed at presenting the most important techniques and tools to build fully functional reconfigurable systems which exploit the raw capabilities delivered by these devices.
Xilinx Inc. has recently announced the integration of DPR design tools into its design framework, making simpler the development of reconfigurable systems and supporting all of its FPGA families [Xilinx Inc., 2012b]. Xilinx’s objective is to make DPR mainstream, promoting its adoption by commercial applications in medium-term. Before this, only an early access to DPR was available, mainly for the Universities and the research community, without any guarantees. Xilinx specifies two design flows: difference-based reconfiguration and module-based reconfiguration, which are described in sections 2.4.1 and 2.4.2, respectively.

However, Xilinx DPR design tools offer limited support for the vast capabilities that partial reconfiguration enables. The reality is that nowadays Xilinx DPR-enabled FPGAs pose much more power than what it is exploitable by using standard DPR design tools. As a result, DPR is currently a hot research topic with innumerable research projects going on.

In the past, Xilinx provided a Java-based library, JBits, to control the configuration bits of Virtex FPGAs [Guccione and Levi, 1999]. JBits translated the high-level configuration changes realised by the designer into the appropriate modification of the configuration bits in the bitstream. Nevertheless, Virtex-4 FPGAs are not supported by JBits and hence, the fine-grained access to Virtex-4 bitstream has not been possible for ages, except to those bits whose function has been previously identified by reverse-engineering (e.g., [Upegui and Sanchez, 2006, Kepa et al., 2009, Bergeron et al., 2007, Note and Rannaud, 2008, Suris et al., 2008a, Korf et al., 2011]).

It has not been until the last two years that the gap left by JBits has been filled. Torc [Steiner et al., 2011] and RapidSmith [Lavin et al., 2011] are two open source tool suites, both based on XDL, which provide a C++ and Java-based interface, respectively, to access (with restrictions) the fine-grained configuration information of modern FPGAs.

### 2.4.1 Difference-based Reconfiguration

Difference-based reconfiguration was the first and most direct way to make use of DPR. In difference-based reconfiguration only small configuration changes are done in order to modify the behaviour of the circuit (e.g., LUT equations, BRAM content, etc.) [Eto, E., 2007]. These changes are typically done using Xilinx FPGA_Editor tool. The reference and the modified design files are processed by Xilinx Bitgen tool to create a partial bitstream which ex-
clusively contains the frames that are different in both designs, resulting in faster reconfiguration times. To do so, "-r" switch must be enabled when executing Bitgen.

An example of an application which uses difference-based reconfiguration is *dynamic data-folding*, where the coefficients that direct the operation of a circuit are reconfigured on-the-fly [Foulk, 1993, Wirthlin and Hutchings, 1997, McGregor and Lysaght, 1999, Al Farisi et al., 2011].

### 2.4.2 Module-based Reconfiguration

In module-based reconfiguration, the FPGA is divided into two regions: the *Partially Reconfigurable Region (PRR)*, whose resources are used to implement different functionalities at different times; and the *Static Region (SR)*, whose resources are used to implement the same functionality at all times; e.g., the *reconfiguration controller* which controls the reconfiguration process by driving the ICAP [Blodget et al., 2003]. As shown in Figure 2.30, module-based reconfiguration allows to allocate a battery of different *Reconfigurable Modules (RMs)* on the same PRR at different times. In contrast with difference-based reconfiguration, where only 2 different configuration states are possible, in module-based reconfiguration there is no limitation in the number of configuration states. On the other hand, the partial bitstreams generated for the RMs contain all the frames in the PRR where they are to be configured, resulting in higher reconfiguration times than difference-based reconfiguration.

With the objective of making all of the RMs to be allocated on the same PRR pin compatible with each other, Xilinx states that all their connections, except the clock signal, must pass through *Bus Macros (BMs)*. Indeed, the clock signal is delivered to the resources through the dedicated clock-tree. BMs are pre-routed CLB-based bridges between the two sides of the PRR, providing a means of locking the routing (see Figure 2.31). Xilinx provides BMs as hard-macros, allowing different options for them. The signals passing through BMs can be registered (the longest path is reduced) or not, and they can go in the four directions, from left-to-right, right-to-left, top-to-bottom or bottom-to-top. Thus, by combining BMs it is possible to exploit the 4 sides of slot boundaries to pass signals through. BMs can include enable control logic to prevent unpredictable values in the signals while the PRR is reconfigured. Two different versions of the BMs are provided with different physical widths: 2 CLBs
for the narrow version and 4 CLBs for the wide version. However, BMs consume extra logic resources and provoke a noticeable impact on the timing of the design.

The current trend towards easing the development of partially reconfigurable systems includes the automatic generation of BMs based on the signal interface of the RMs [Claus et al., 2007, Korf et al., 2011] and the automatic placement of the BMs to optimise the timing of the design [Carver et al., 2008]. Indeed, in the latest release of Xilinx DPR design tools, BMs are replaced with proxy logic that is automatically inserted by Xilinx design tools. In [Koch et al., 2010b], an alternative method to communicate the SR and the PRR is proposed that removes the logic overheads introduced by BMs. The authors propose to lock the routing using the inner routing resources of the FPGA.

The use of BMs placed at fixed positions results in fixed boundaries for the PRR. The PRR is thus turned into a set of reconfigurable slots whose size and shape is defined by the system designer at design time (e.g., [Bobda et al., 2005b]). However, as this size must be chosen based on the largest task, the rest of the RMs are unnecessarily enlarged within the slot, and consequently, slot-based reconfiguration results in inefficient resource exploitation. Furthermore, the number of RMs which can be concurrently allocated on the device is limited by the amount of reconfigurable slots included in the system.
Despite the internal signals of the RMs do not cross the boundaries defined for the reconfigurable slots where they are allocated, the physical lines used to route these signals interconnect different FPGA resources which might belong to distinct reconfigurable slots. As shown in Figure 2.14, up to 3 FPGA resources are interconnected by means of double and hex lines, while long lines interconnect all of the resources contained in an FPGA row or column. Note that the PIPs associated to lines which do not interconnect with other resources in the same reconfigurable slot are never used. This can be seen in Figure 2.32a, where PIP3 is never activated as it does not allow to interconnect with any other resource in the reconfigurable slot (B), while PIP1 and PIP2 can be activated to route a signal between two resources lying in the same reconfigurable slot (A). In order to avoid routing conflicts, the lines which interconnect two or more resources lying in different slots are prevented from being used in RMs (e.g., long lines). In Figure 2.32b, a shortcut is provoked when signal A and signal B are not at the same logic level as they share the same physical long line.

HDLs continue to be valid to describe partially reconfigurable systems. The top-level hierarchy HDL file includes all global logic that must be static, such as I/Os, global clocks, DCMs and the ICAP itself. It is strictly prohibited to use global logic inside the PRR. In addition, the top file includes all the static logic and the instantiation of the reconfigurable slots, which are considered “black-boxes”, with the BMs interfacing all their I/O signals.
Chapter 2- Introduction to FPGAs and Dynamic Partial Reconfiguration

(a) Double and hex lines
(b) Long lines

Figure 2.32: Implications of programmable route lines in module-based reconfiguration

Figure 2.33 shows Xilinx design flow for module-based reconfiguration [Xilinx Inc., 2012b]. The static logic is first synthesised. While the static logic is placed outside from the PRR, the static routes to connect this logic are not constrained and therefore, they can cross across the PRR. Indeed, in order to avoid conflicts, the routing resources used to route static signals inside the PRR are not considered when synthesising each of the RMs. Note that RMs are self-contained in the PRRs where they are to be allocated; i.e., neither their logic nor their routes go outside PRR boundaries. Finally, the static and partial bitstreams are merged to ensure the persistency of the configuration data in all of the possible configurations for the PRR, guaranteeing the integrity of the static signals crossing the PRR. The main problem of this flow is that generated partial bitstreams are location specific; i.e., multiple partial bitstreams must be generated for a single RM to be allocated on multiple locations within the PRR, each accounting for the existing static routes in the corresponding location.

Figure 2.33: Xilinx design flow for module-based partial reconfiguration
To help in the floor-planning of the reconfigurable system (i.e., determining the shape, size and placement of the reconfigurable slots as well as the location of BMs), Xilinx has developed a graphical interface tool named PlanAhead [Dorairaj et al., 2005]. PlanAhead translates designer’s specifications into placement constraints which are used in the PAR process. Namely, AREA_GROUP constraints prevent logic in the base design from being merged with the logic in the RMs; AREA_GROUP_RANGE constraints set the placement and shape of the RMs/slots; and MODE constraints define a module as reconfigurable. Additionally, PlanAhead checks whether all the design rules are met in the partially reconfigurable system. To implement the system, PlanAhead invokes XST following the reconfiguration flow shown in Figure 2.33.

Figure 2.34 shows an exemplary partially reconfigurable TMR system (taken from [Iturbe et al., 2009]), where three of the slots in the PRR allocate an identical RM and the remaining slot allocates the majority voter. In this system DPR is used to correct soft-errors, which are detected by the majority voter when any of the three slots gives a distinct output. This triggers the scrubbing of the corrupted slot, which consists in reconfiguring it with the correct RM configuration. Figure 2.34a shows the floor-planning of the system in PlanAhead and Listings 2.5 includes the resulting placement constraints for that floor-planning. In Figure 2.34c, note the large amount of static routes crossing the PRR, despite the SR includes exclusively the reconfiguration controller.

2.4.3 Bitstream Relocation: Slotless Reconfiguration

Bitstream relocation is a relatively new trend that explores the possibility of exploiting the regular architecture of Xilinx FPGAs to use the same partial bitstreams in different locations. Indeed, based on the relation between the configuration memory map and the configured physical resources, a partial bitstream can be relocated to any arbitrary position within the FPGA by simply “shifting” its configuration data within this memory. Partial bitstream relocation alleviates the storage requirements in the system as it is not necessary to keep multiple configuration information of the same module for each position to be allocated.

There are three conditions to meet when relocating a module.

1) The target position must be identical to the source position in the type and arrangement of resources as well as communication interfaces.
Listing 2.5: Placement constraints generated by PlanAhead for the floor-planning shown in Figure 2.34a

```
# Slot1: TMR Module 1
INST "TMR1" AREA_GROUP = "pblock_TMR1";
AREA_GROUP "pblock_TMR1" RANGE=SLICE_X26Y96:SLICE_X35Y127;
AREA_GROUP "pblock_TMR1" RANGE=DSP48_X0Y24:DSP48_X0Y31;
AREA_GROUP "pblock_TMR1" MODE=RECONFIG;

# Slot2: TMR Module 2
INST "TMR2" AREA_GROUP = "pblock_TMR2";
AREA_GROUP "pblock_TMR2" RANGE=SLICE_X26Y64:SLICE_X35Y95;
AREA_GROUP "pblock_TMR2" RANGE=DSP48_X0Y16:DSP48_X0Y23;
AREA_GROUP "pblock_TMR2" MODE=RECONFIG;

# Slot3: TMR Module 3
INST "TMR3" AREA_GROUP = "pblock_TMR3";
AREA_GROUP "pblock_TMR3" RANGE=SLICE_X26Y32:SLICE_X35Y63;
AREA_GROUP "pblock_TMR3" RANGE=DSP48_X0Y8:DSP48_X0Y15;
AREA_GROUP "pblock_TMR3" MODE=RECONFIG;

# Slot4: Majority Voter
INST "voter" AREA_GROUP = "pblock_voter";
AREA_GROUP "pblock_voter" RANGE=SLICE_X26Y0:SLICE_X35Y31;
AREA_GROUP "pblock_voter" RANGE=DSP48_X0Y0:DSP48_X0Y7;
AREA_GROUP "pblock_voter" MODE=RECONFIG;

# Bus-Macroses
INST "bm_out_data_TMR1" LOC = SLICE_X24Y116;
INST "bm_ctrl_TMR1" LOC = SLICE_X34Y122;
INST "bm_in_data_TMR1" LOC = SLICE_X34Y108;
INST "bm_out_data_TMR2" LOC = SLICE_X24Y48;
INST "bm_ctrl_TMR2" LOC = SLICE_X34Y52;
INST "bm_in_data_TMR2" LOC = SLICE_X34Y48;
INST "bm_out_data_TMR3" LOC = SLICE_X24Y48;
INST "bm_ctrl_TMR3" LOC = SLICE_X34Y52;
INST "bm_in_data_TMR3" LOC = SLICE_X34Y48;
INST "bm_in_data_voter1" LOC = SLICE_X34Y12;
INST "bm_in_data_voter2" LOC = SLICE_X24Y12;
INST "bm_in_data_voter3" LOC = SLICE_X24Y4;
INST "bm_voter_out_data" LOC = SLICE_X34Y20;
INST "bm_voter_out_error_id" LOC = SLICE_X34Y6;
```
In [Koester et al., 2009] the authors describe a design method for selecting a synthesis region for the relocatable modules with the objective of optimising their placement at runtime and, in [Becker et al., 2010], the authors describe an automated method for finding feasible relocation positions for the modules.

In [Becker et al., 2007] the authors explain the basis for dealing with the heterogeneous resources incorporated in last generation FPGAs. They succeed in allocating a module in a target position with different types of resources to the original region for which it was synthesised (see Figure 2.40). This can be done because the routing structure is regular along the chip: the routing resources associated to the BRAM columns in the original position are the same as the resources associated to the BRAM and DSP48 columns in the target position. However, note that in this figure that it is not possible to shift the module one more column to the left or right of the target position, as neither the relative location of the heterogeneous resource columns nor the position of communication interfaces would match. The implications of having to preserve the communication interfaces in the target position are discussed in section 2.4.4.

![Figure 2.35: Limitations of partial bitstream relocation due to heterogeneous resources](image)

Finally, note that the vertical allocatability of the modules is constrained by the amount of existing clock regions in the device. Currently the feasibility of “shifting the configura-
tion bits within a configuration frame” is not proved and thus, the vertical placement of a relocatable module within a clock region cannot be modified at runtime.

2) **The static signals crossing the target position must be preserved.** As Xilinx tools do not support relocation yet, the merge between static routes and relocated partial bitstreams must be made manually when downloading the bitstream to the target position.

In [Sedcole et al., 2006] the authors present the Merge Dynamic Reconfiguration (MDR) method, which extends the RMW technique to preserve the static routes in the target position; i.e., the configuration frames in the target position are read-back and XORed with the frames of the partial bitstream prior to being written-back in the configuration memory (see Figure 2.36). Likewise, the same XOR operation is used to remove the module from the source position; i.e., \( a \oplus b \oplus b = a \).

![Figure 2.36: Preserving the static signals when relocating a module](image)

In the worst-case a module relocation could be prevented if the latter needs to use the same routing resources already assigned to the crossing static signals in the target position. Worse still, note that due to the way SBs and CBs are implemented, where each PIP is connected to multiple other PIPs, it is not immediate to detect routing conflicts. As shown in Figure 2.37, two routes which do not have any PIP in common can provoke a conflict if their PIPs share common interconnections in the SB/CB [Bellato et al., 2004]. Therefore, a thorough knowledge of device's routing-related configuration information is mandatory to detect routing conflicts.
To avoid routing-related issues, certain routing resources can be exclusively reserved for static routing, and prohibited to be used by the RMs. In [Sedcole et al., 2006] a percentage of the hex lines and all of the long lines in the FPGA are allocated to static routes, arguing that modules do not need these resources due to the local nature of their interconnections.

To do so, a specific script which runs on top of Xilinx design tools is created. The script loads a placed and routed module design file, unroutes all signals which use illegal PIPs, reroutes these signals using valid PIPs only, and saves the resulting design to a file [Sedcole, 2006]. Alternatively, a subset of the routing resources within a region of the FPGA can be prohibited by using the blocker macros presented in [Koch, 2009]. Similarly, in [Sohanghpurwala et al., 2011], all routing wires that have either a source or destination within the PRR are blocked by inserting an “anti-core” (see Figure 2.38c). However, these approaches usually result in longer critical paths (i.e., lower achievable frequency), as the usable routes are constrained.

There are some constraints provided by Xilinx to prevent static routes from crossing through the PRR. A constraint called ALLOW_ROUTING_IN_DYNAMIC_AREA is reported in [Montminy et al., 2007]. However, the authors have no success in synthesising the reconfigurable system when this constraint is used. Besides, no other reference to this constraint was found. In fact, the Xilinx standard constraint to prohibit the use of routing resources inside the PRR is ROUTING = CLOSED, which has been recently replaced by PRIVATE = ROUTE. However, there are some problems reported when using the former constraint in Virtex-4 FPGAs [Carver et al., 2008] and the latter does not work at all in newer FPGA families when using the latest releases of Xilinx design tools. This is due to the BMs employed, which have inputs that reside within the PRR and therefore, the router must enter the region to connect to them [Asgar Sohanghpurwala, 2010].
Figure 2.38: Design process to prevent static routes from crossing through the PRR [Asgar Sohanghpurwala, 2010]
3) **The clock signal must feed all resources in the target position.** As the synthesiser does not route the clock to the resources which are not used in the original configuration, it may happen that circuitry is relocated at runtime to any of these unclocked resources. This is not a problem in traditional module-based reconfiguration because the circuitry is allocated only to reconfigurable slots.

In [Flynn et al., 2009], regional clock buffers are used to deliver the clock signal to the resources. Basically, this approach includes the BUFRs as a component of the RMs. Indeed, Xilinx provides practical information on the use of BUFRs in a partially reconfigurable design in [Eto, 2008]. Despite being functional, this solution is limited. First, as the partial bitstream of the RMs include the configuration of the BUFRs as well, each clock region can host only those RMs running at the same clock frequency. Every time a new RM is configured in a clock region the configuration of the BUFR in that region is overwritten. Second, as the RMs can be allocated only to the positions where BUFRs are located, they cannot be horizontally shifted inside a clock region. Finally, as the RMs are fed by a BUFR, their height must be no greater than 3 clock regions.

A more advanced approach is found in [Schuck et al., 2011], where the authors explain the basis to dynamically route clock signals to the RMs in Virtex-II devices. However, as these FPGAs do not include a branched clock-tree, it is significantly difficult to distribute the clock signal along the device.

In *slotless reconfigurable systems*, the PRR becomes a nearly homogeneous resource (only constrained by the resource arrangement) to be shared by multiple modules with different sizes and shapes which fit the exact amount of required resources. Therefore, the number of concurrent modules is limited only by the amount of resources and BMs and not by the amount of reconfigurable slots defined at design phase. This results in a more efficient use of FPGA’s resources.

Bitstream relocation was initially intended to be done offline. This is the case of PARBIT [Horta et al., 2002] and BITPOS [Krasteva et al., 2005], which generated a set of relocated partial bitstreams using as input the files generated by Xilinx tools. The relocated bitstreams were used to hot-swap application-specific RMs at runtime. While PARBIT ex-
clusively relocated CLBs, BITPOS could relocate memory and multiplier blocks as well, thus supporting the Virtex-II heterogeneous architecture.

The next objective was to execute the relocation process online. This is the case of XPART [Blodget et al., 2003], an API provided by Xilinx Inc. with methods for bitstream relocation, and pBITPOS [Krasteva et al., 2006], a lighter version of BITPOS especially intended to run on an embedded processor with limited computing power. Other approaches that use an embedded processor to perform the relocation can be found in [Sedcole et al., 2006, Becker et al., 2007].

To speed-up the relocation process, bitstream filters have been proposed. Bitstream filters are hardware solutions that transform the frame addresses in the bitstream according to the target position during the regular download process. REPLICA [Kalte et al., 2005] was the first filter with solely support for CLB relocation, while REPLICA2Pro [Kalte and Porrmann, 2006] included support for dealing with memory and multiplier columns. Finally, BiRF was a bitstream filter developed for tile-based FPGA architectures such as Virtex-4 [Corbetta et al., 2009]. The general architecture of these filters include three main components: (1) an FSM, which parses the original bitstream to detect the frame addresses and the CRC commands, (2) a relocation controller, which generates the new frame addresses in the relocated position, and (3) a CRC block, which computes a new CRC value for the relocated bitstream.

### 2.4.4 Reconfigurable Inter-Connectivity

RMs must be provided with inter-connectivity in the positions where they are moved to. Otherwise, they will be isolated and will not be able to process any data.

Ideally, a reconfigurable system should be able to online route the relocated modules. Technically, it is possible to directly use FPGAs routing resources by activating the appropriate PIPs (see Figure 2.39a). However, this is a long time consuming task, due to the vast amount of routing resources available in modern FPGAs [Huang, 2004]. In addition, its success is constrained by the closed nature of the bitstream format. JRoute [Keller, 2000] and ADB [Steiner and Athanas, 2004] were two tools that operated on the configuration layer of the FPGA using JBits. However, as the latter does not support modern Xilinx architectures, both tools are nowadays deprecated. The aforementioned Torc tool, which does support
modern Xilinx FPGAs, includes a router which allows to use the PIPs even to create a net with specific different geometric shapes [Couch and Athanas, 2011]. Nevertheless, as far as we know there is no reported use of Torc in online scenarios.

In [Suris et al., 2008a], the authors describe a router that creates Wires-on-Demand (WoD) using the PIPs in the new Xilinx FPGAs. To reduce the computing requirements of the router, and thus enable its execution on an embedded processor, it only uses double and direct lines. The router consults a routing resource database, obtained from XDL files, which is kept small by exploiting the repetitiveness of the routing fabric. The authors report that, when the WoD router is executed in a desktop computer, the amount of used memory is reduced by three orders of magnitude and the execution speed is up to four orders of magnitude faster compared with proprietary tools. Although it is reported the use of the WoD router in a Software Defined Radio (SDR) prototype [Suris et al., 2008b], no performance results are provided when running on an embedded processor. Note that the execution time of a similar routing algorithm is measured in the range of tens of seconds when executed in an embedded PowerPC processor [Silva and Ferreira, 2012]. A similar approach for online routing is presented in [Koch et al., 2010a]. The main contribution of this work is to provide the router with capability to check that online created routes meet the timing constraints of the system; i.e., the propagation delay is smaller than the used clock period.

An alternative to directly managing the PIPs is to use the logic resources in the fabric to route signals through. In [Hubner et al., 2006], online routing is based on basic CLB-based Routing Primitives (RPs) which implement both vertical routing connections and module interfaces. RPs are concatenated in a slotted column to create the desired vertical communication channel. As shown in Figure 2.39b, the communication infrastructure is completed with a horizontal bus hard-macro to connect the slotted columns among them. In [Shayani et al., 2008], the authors extend this idea by using relocatable RPs which can be freely placed around the modules to form arbitrary communication channels (see Figure 2.39c).

However, using logic resources for routing purposes brings some drawbacks with respect to performance and efficiency. First, the successive concatenation of combinatorial routing logic may increase the longest path in the system, resulting in timing violations. Despite this can be circumvented by registering the signal through flip-flops in the RPs, a variable communication latency will continue to exist, making necessary a handshake protocol.
Second, the exploitable logic capacity of the FPGA is significantly reduced, not only due to the direct usage of logic resources for routing, but also due to the fragmentation provoked by the routes.

In [Silva and Ferreira, 2008], the authors present a layered system where the modules allocated in each layer can be connected only to modules in an adjacent layer. In this context, inter-module connections are limited and selected from a set of predetermined routes which have been envisioned at design time (see Figure 2.39d). The predetermined routes can be seen as larger communication primitives which use the resources in the fabric more efficiently; i.e., logic is included only at the input and output and internally fabric's routing resources are exclusively used. Despite it is not reported by the authors, the modules could be freely placed in the device if: (1) there is any predefined route to connect them in their target locations and, (2) the previously existing routes are preserved when configuring a new module. The drawback of this technique is the reduced reusability of the predefined routes which results in increased storage requirements; i.e., every connection between two different positions defines a route to be stored.

Other research efforts rely on fixed communication infrastructures which can be rapidly adapted to serve the changeable communication requirements of the relocatable modules. Rapid adaptation is achieved at the price of consuming a considerable part of the resources in the chip. Note that a design flow to reduce the resource requirements is presented in [Korf et al., 2011]. This flow, named DHarMa, exploits the regularity of resources in the FPGA chips to implement homogeneous communication infrastructures with lower area overheads.

In [Upegui and Sanchez, 2006], LUT-based multiplexers are used to implement a reconfigurable crossbar. By using difference-based reconfiguration, the truth table of the multiplexers can be modified to create online connections among the modules. This permits to reduce the logic overheads by removing the multiplexer selection logic. In [Ahmadinia et al., 2005], the authors present a Reconfigurable Multiple Bus (RMB) consisting of an array of parallel bus segments which interconnect a set of reconfigurable slots. The routes among the slots are dynamically created by linking several segments together at the cross-points as shown in Figure 2.40a. Each cross-point includes a controller to process the switching requests coming from the slots or from other cross-points when a route is being established.
Note that this architecture mimics at a higher scale the underlying routing structure of the FPGA fabric, where the low-level switch boxes are analogue to the high-level cross-points. A similar communication infrastructure is used in VAPRES, but the switching is controlled by an embedded processor \cite{Jara-Berrocal and Gordon-Ross, 2010}. In \cite{Koch et al., 2008b} the authors describe a set of techniques to enhance the flexibility of integrating reconfigurable modules efficiently into a reconfigurable bus, ReCoBus. To reduce the amount of used wires, the ReCoBus is time-shared by a potentially large number of modules.

All of the aforementioned approaches are aimed at creating physical connections among the communicating modules (circuit switching), but there are other research efforts which use logic connections instead (packet switching). The Sonic-on-a-Chip architecture retains some interesting implementation features \cite{Sedcole et al., 2007}. This architecture consists of an array of homogeneous slots interconnected by means of a bus that transports the data packets. The modules connect to the bus at predetermined fixed locations and, where slots...
are unused or modules occupy more than one slot, the bus connection points are inactive. As these points are located in a regular fashion along the chip, the modules can be freely shifted by a minimum number of slots in any direction (see Figure 2.40b). In addition, each module includes sockets for ChainBus connections to their immediate neighbours.

The best example of logical connections is found in NoCs. Most of the NoCs used in reconfigurable systems consist on a fixed infrastructure where the routing tables are adjusted each time a module is reconfigured (e.g., [Devaux et al., 2010, Palesi et al., 2007]). Other NoCs make active use of partial reconfiguration to adapt their topology to emerging needs over time. An early example of such NoCs, called DyNoC, can be found in [Bobda et al., 2005a]. As shown in Figure 2.40c, DyNoC routers are reused by a module as part of its logic, and when the module is deallocated, the original routers are again restored. Hence, the topology of DyNoC varies according to the modules connected to it at any time. CoNoChi goes one step further as its complete topology can be adapted to the number and location of the modules connected to it [Pionteck et al., 2006]; i.e., new network interfaces and routers can be created on-the-fly. To do so, different types of RMs are used, namely routers, horizontal routing, and vertical routing blocks (see Figure 2.40d). The novelty of ReNoC is that it combines logical packet-switching with physical circuit-switching within the same NoC [Stensgaard and Sparso, 2008]. Hence, the topology of the NoC can be adapted based on the communication demands of the modules; i.e., more physical connections can be created to deal with traffic congestion in a given segment (see Figure 2.40e).

To circumvent the overheads introduced by the fixed communication infrastructures, an innovative solution is proposed in [Shelburne et al., 2008]. In this work a NoC is emulated by harnessing the reconfiguration mechanism of the FPGA: data is read-back directly from the network interface of a sender module and written-back to the network interface of a receiver module. That is, ‘virtual’ communication channels are established between the communicating modules through the configuration layer of the FPGA. While this method leads to very little logic overheads, it does not support concurrent communications and incurs reconfiguration overheads due to data relocation [Sander et al., 2008]. Note that this communication method was originally proposed by Brebner et al. in [Brebner and Danlin, 1998], albeit using early FPGA technology.
Figure 2.40: Communication infrastructures in RC systems
2.4.5 From Reconfigurable Platform-based Design to ROS-based Design

The main barrier for adopting DPR in industrial applications seems to be the lack of tools, methodologies and runtime support [Torresen and Koch, 2011].

The previously presented Xilinx PlanAhead tool considerably simplifies the development of slotted partially reconfigurable systems, especially the floor-planning, but it cannot deal with slotless reconfiguration and lacks of support for inter-module communications. It only provides point-to-point physical communication between the static and the partially reconfigurable regions using BMs. OpenPR gives a similar functionality as PlanAhead but offers support for slotless reconfiguration as well [Sohanghpurwala et al., 2011]. However, to enable the fast and automatic development of reconfigurable systems a capable reconfigurable design platform must be created. Besides including the design automation capabilities available in standard platforms (see section 2.1.2), the reconfigurable platform should add the aforementioned two capabilities among others (i.e., support for inter-module communications and floor-planning).

ReCoBus-Builder (now called GoAhead) is a design platform with support for slotless reconfiguration that uses the previously presented ReCoBus as communication infrastructure [Koch et al., 2008a]. Based on the specifications given by the designer, the tool generates a hard-macro containing the static logic and routing of the ReCoBus that provides the communication among the reconfigurable modules. In addition, it generates the HDL files that instantiate the reconfigurable modules following the ReCoBus interface protocol specification. These files are to be synthesised using the standard XST. To do the floorplanning of the system ReCoBus-Builder offers a practical graphical interface to the designer. Unlike PlanAhead, ReCoBus-Builder does not use placement constraints to assign the resources to the SR and PRR, instead it uses the so-called blocker macros to prohibit the usage of both logic and routing resources outside each of the regions. Namely, when synthesising the static part, the area assigned to the PRR is prohibited, and when synthesising each of the RMs, blockers ensure they do not use any logic and routing resources outside the specified module bounding box. ReCoBus-Builder also generates a behavioural HDL description of the ReCoBus to enable the simulation of the whole system. However, it does not include any high-level simulation tool to simulate the behaviour of the system.
DRAGOON is an environment which allows for the automatic generation of a NoC adapted to the needs of a given reconfigurable system [Devaux et al., 2010]. However, it cannot be considered a complete design platform as it does not assist the designer during the floor-planning. On the other hand, DRAGOON permits to simulate the NoC to evaluate its behaviour when dealing with different traffic situations.

Despite the fact that traditional RTL simulation tools are appropriate for (individually and separately) debugging each RM and the static configuration controller itself, there is no suitable simulation tool to help in the system integration phase. The most common method to simulate DPR consists in MUX-like functionality to switch between active RMs (e.g., [Lysaght and Stockwood, 1996, Luk et al., 1997, Raabe et al., 2008]). This permits to model module swapping process and hence, it is suitable for high-level TLM simulation. Nevertheless, the hardware-related nature of DPR and the lack of any established reconfigurable platform make essential to simulate the reconfiguration process at RTL level as well [Gong and Diessel, 2011a]. Indeed, most of the logic to control the reconfiguration, including the bitstream data-path and the interfaces of the RMs, is currently defined by designers and hence is prone to error. Up to date, only the Dynamic Circuit Switching (DCS) framework [Lysaght and Stockwood, 1996, Robertson et al., 2002] approaches to RTL simulation of DPR by adding some artifacts to the HDL description of the partially reconfigurable system (e.g., a delay to model the reconfiguration time). However, as these artifacts cannot be reused in different designs, DCS is not suitable for platform-based design. In [Gong and Diessel, 2011b], the authors introduce ReSim, a generic and reusable simulation layer to capture the physical-dependent details of DPR. Furthermore, ReSim permits to balance between accuracy and physical independence to help the designer at both RTL and TLM level simulation. In [Gong and Diessel, 2012], an extension to simulate the transfer of state between the application logic and the configuration port (e.g., GCAPTURE and GRESTORE commands) is proposed.

Besides developing new tools and methodologies to ease the design of reconfigurable applications, runtime support must be provided to execute them. A proposal that is gaining momentum in the last years is to implement OS-like support for reconfigurable hardware so that all common operations (e.g., inter-task communications and synchronisation), could
be easily and safely carried out. This would also make reconfigurable systems more predictable and amenable to be modelled using TLMs, thus easing debugging and simulation.

2.5 Chapter Conclusion

The dawn of 21st Century has brought a real revolution in reconfigurable hardware. FPGAs have turned into sophisticated, flexible and extremely advanced compute fabrics which can change their own functionality on-the-fly (i.e., self-reconfiguration). As a result, the necessity for tools and methods to exploit the advanced computation capabilities delivered by current devices is now more potent than any time before.

Currently designers face a complex design flow and a set of complicate implementation tools when designing their reconfigurable application. However, this should not be in this way: designers should deal with application details rather than with implementation details. Furthermore, there is a need for additional support to explore other modes of partial reconfiguration, currently not supported by Xilinx tools (e.g., hardware relocation). More significantly, there is little or no runtime support for executing reconfigurable applications. This situation is similar in concept to what occurred during the infancy of electronic computers, when dealing with machine's hardware was direct and messy. Now, as then, the use of an OS (ROS in the case of FPGAs) is an attractive proposition to make partially reconfigurable FPGAs “user friendly” and more productive.
In the early days of computing, electronic computers were operated manually by users. Each user was allocated a time slot, which was mostly spent setting up the equipment to do his/her computation, resulting in huge economical losses. To increase efficiency, computers were enabled with the capability to manage their own workload by means of a software Operating System (OS). The OS provided a software platform on top of which user application programs could run. Besides, the OS alleviated application programmers from having to manage computer resources at a low-level. In order to deal with the slow card readers and line printers in that age, the OS scheduled user applications at different times. A user application waiting for a response from a slow peripheral was temporarily stopped and another user application which already had received a response got its turn to be executed meanwhile. Overall, the OS was responsible for the management and coordination of the execution of user applications and of sharing limited computer resources.

The innovation brought about by Dynamic Partial Reconfiguration (DPR) of FPGAs set the technological basis for the development of an ambitious computing paradigm, called Reconfigurable Computing (RC), that had not been able to make the transition from theory to practice since it had been proposed by Dr. Estrin as early as 1960 [Estrin, 1960]. In general, the RC paradigm targeted a computer with a dynamically adaptable architecture to

---

Part of this chapter has been accepted for publication in [Iturbe et al., 2013a, Iturbe et al., 2013b].
match the needs of the computation to be performed. DPR fulfilled this requirement as it allowed for continuous processing of data as hardware modules were dynamically allocated on FPGA, executed and finally replaced by other modules.

Inspired by software, Dr. Brebner from the University of Edinburgh was among the first researchers to propose a Reconfigurable Operating System (ROS) to automat many operations of a reconfigurable computer [Brebner, 1996]. As in a software OS, a ROS is to coordinate and manage a potentially high number of computations, which could even be launched by different users, and compete for access to a limited amount of resources on FPGA. Moreover, as occurred in software, the ROS is expected to increase the user base of FPGAs by hiding the complex hardware-related issues of these devices.

This chapter reviews the main concepts related to RC, and describes the most important research efforts conducted up to date to build a ROS. First, the chapter presents the concepts related to software OS as some of these concepts continue to be valid in the scope of a ROS. Finally, the chapter briefly discusses the role that ROSs can play in the context of a software abstraction technology that has gained importance in the last years: Hypervisors.

### 3.1 (Software) OS Features

The *kernel* is the central part of an OS. It directly controls the underlying hardware in the machine to provide a clean and uniform interface to it. Indeed, the kernel (1) decides when and how long each computation can make use of the CPU, (2) manages memory and peripheral access, and (3) implements a set of mechanisms to support the development and execution of complex applications. In order to execute an application, the OS kernel assigns separate memory space and other hardware resources to each of the tasks, loads tasks’ code into these memory positions, and coordinates their interaction, as well as with the user if necessary. In traditional mono-core processors, only one task can run at a time on the CPU, and therefore the different tasks in the application must be conveniently scheduled over time. *Context switch* or *task preemption* refers to the process of reassigning the CPU from one task to another. When context switches occur frequently enough the illusion of parallelism is achieved (i.e., multitasking). In multi-processor systems, multitasking allows more tasks to be run than there are available CPUs.
As depicted in Figure 3.1, a software OS typically has a ring architecture. Each ring implements a different set of features and services, which can be invoked through a set of predefined system calls. The most important features and services offered by current OS are briefly summarised in this section.

![Figure 3.1: Software OS ring architecture](image)

**Memory Management** Memory is carefully managed by the OS kernel to prevent interference when multiple tasks access the same locations. Modern CPUs include hardware support, namely a Memory Management Unit (MMU), to assist the OS to limit access to memory by the tasks. More specifically, the memory positions allowed to be accessed by user tasks are specified by means of some registers in the MMU. Unrestricted access to memory is only allowed when running in privileged mode, which typically is only available for the OS kernel itself. Any attempt to access a non-allowed memory position is detected and handled by the OS.

**Virtual Memory** Virtual memory refers to the act of managing all of the heterogeneous data storage resources available on the machine (e.g., hard-disk, RAM) as if they were a single, homogeneous and directly addressable read/write memory. Therefore, a contiguous range of virtual addresses are mapped to several non-contiguous blocks of physical memory, with the MMU responsible for translating virtual memory addresses to physical addresses. Furthermore, virtual memory provides a way to give the impression that there is more memory than physically available in the system and allows for each task to have its own address space. In conclusion, virtual memory makes application programming easier by freeing the developer from having to deal with the fragmentation of physical memory, management of the memory hierarchy (e.g., control data transfers between main memory and caches), and relocation of tasks’ code.

**File System** A file system is a means to organise the data managed by the application. It provides functions to store, retrieve and update data on the memory.
Inter-Task Communications and Synchronisation  Inter-task communication refers to the mechanism that allows for the exchange of data between tasks. Typically, this requires synchronising the tasks when accessing shared memory locations. The most typical synchronisation mechanism is a *semaphore* that allows *taking* and *releasing* shared resources by tasks \[\text{[Dijkstra, 2002]}\]. Besides, data can be passed from task to task by using file descriptors (e.g., pipes).

Device Drivers  Device drivers refer to software that allows interaction with a specific hardware device, abstracting it to a user. The driver is aware of all of the particularities of the hardware device in order to manage it based on hardware-independent directives received from the OS. Typically, drivers refer to input/output devices (e.g., keyboard, video monitor).

Resource Allocation  The OS coordinates access to limited system resources, including CPU time, memory and peripherals.

Interrupt  Interrupts are the easiest and fastest mechanism to indicate the need for attention to the OS. Interrupts can be hardware, e.g., a timer sends interrupts at periodic intervals, or software, e.g., system calls. Each interrupt is attended by a specific Interrupt Service Routine (ISR), which essentially updates the state of the system, allowing it to quickly respond to upcoming events.

Networking  Modern OSs offer support to access remote devices and resources through a data network (e.g., using sockets).

3.1.1 Monolithic Kernels and Microkernels

There are two different concepts of kernel: *monolithic kernel* and *microkernel* (uK). While all of the OS services implemented by a monolithic kernel are embedded into a single operating system file, a microkernel includes the bare minimum functionality to support the OS services which run on top of it (see Figure 3.2). The idea here is to separate the core kernel services from the remaining more complex and “baroque” services. The latter are indeed implemented as different *Servers*, with the microkernel responsible for ensuring coordination and cooperation among them. As a result, microkernels are reduced in size and functionality, reliable, portable, scalable, and easier to extend and maintain than monolithic kernels \[\text{[Liedtke, 1996b]}\].
As an example, the L4 microkernel occupies only 12 KB, provides only three basic abstractions (i.e., task/process, address space and inter-task communication) and seven system calls on top of these abstractions [Liedtke, 1996a]. Being so small, the L4 microkernel can completely reside in a processor’s first level of cache. On the other hand, the size of the compressed file containing the Linux monolithic kernel source was about 18 MB in 1999, with a growth of around 90% in the last 5 years [Godfrey and Tu, 2000].

3.1.2 Real-Time Performance

In today’s world many potentially dangerous pieces of equipment are controlled by OS managing embedded systems, which must fulfil precise timing constraints, usually dictated by events from their environment [Buttazzo, 2004]. This equipment includes cars, trains, airplanes, and nuclear power plants, among others. Furthermore, mainstream application developers are increasingly asking for some real-time features to improve the user experience and to ensure a good Quality of Service (QoS).

The concept of a task that is invoked repeatedly is central to real-time systems. A real-time task \( \tau_i \) is typically modelled by its arrival time or release time \( r_i \), its computing time \( C_i \) and its period \( T_i \) (only valid for periodic tasks). Hence, successive arrival times of a periodic task are \( r_{i,k} = r_{i,0} + kT_i \), where \( r_{i,0} \) is the first release time and \( r_{i,k} \) is the \( k + 1 \) release. On the other hand, a sporadic task is invoked at arbitrary times but with a specified minimum time interval between invocations. The time-constrained processing requirements of a real-time task is modelled by means of a relative deadline \( \text{D}_i \), which represents the maximum acceptable delay for its processing. The relative deadline allows computation of the
absolute deadline for each task arrival \( d_i = r_i + D_i \), as shown in Figure 3.3. Hence, successive absolute deadlines of a periodic task are \( d_{i,k} = r_{i,k} + D_i \). A real-time task is well formed if \( 0 < C_i \leq D_i \leq T_i \).

![Figure 3.3: Real-time periodic (software) task model](image)

Transgression of the absolute deadline of a task causes a timing fault. Depending on the consequences of the timing fault, two classes of real-time tasks are distinguished. A timing fault affecting a soft real-time task makes the performance of the system decrease, but it does not jeopardize its correct behaviour. On the other hand, a timing fault affecting a hard real-time task could cause catastrophic consequences on the system and its environment.

Task activation paradigm is a central aspect in real-time computing. When activities are initiated at pre-defined points in time (i.e., tasks are released at pre-defined instants), the activation paradigm is time-triggered. In this approach a feasible schedule, where all the computation deadlines are met, can be constructed offline based on the anticipated knowledge of the system functioning (e.g., inter-task communications, task allocation). At runtime, a very simple dispatcher executes the scheduling decisions made offline. However, when the real-time system must react to unpredictable events, or in data-dependent computation, the activation paradigm is event-triggered, meaning that tasks must be scheduled online.

In order to achieve the best real-time performance in event-triggered systems, most Real-Time Operating Systems (RTOS) assign a priority to each task, with higher priority tasks always scheduled before lower priority ones. Static priorities are assigned offline, e.g., the task with the shortest relative deadline receives the highest priority in Deadline Monotonic (DM) \([\text{Leung and Whitehead, 1982}]\), while dynamic priorities are assigned online, e.g., the task with the closest absolute deadline receives the highest priority in Earliest Deadline First (EDF) \([\text{Liu and Layland, 1973}]\). Dynamic priorities permit the better exploitation of the processor’s computational power and lead to finer adaptation to unpredictable environments.
On the other hand, static priorities involve less scheduling computation at runtime, thus reducing time overheads.

In preemptive scheduling, a high priority task always preempts any lower priority task that could be executing, while in non-preemptive scheduling, the high priority task does not execute until the task that was executing finishes (see Figure 3.4). In general, preemptive scheduling is less prone to timing faults as it allows for more immediate attention to events (i.e., fast response time). However, it incurs higher time and memory overheads, mainly due to task context switches.

![Preemptive vs. non-preemptive scheduling](image)

**Figure 3.4:** Preemptive vs. non-preemptive scheduling

### 3.2 FPGAs defining Scenario: Reconfigurable Computing

While chapter 2 presented the most important low-level hardware-related particularities of current Xilinx FPGAs, this section is aimed at discussing the implications of FPGAs from a computational viewpoint.

In general terms, an FPGA can be seen as a two-layered architecture, where the functional layer, which contains the physical resources used to perform computation, is controlled by the configuration layer. FPGAs are capable of performing on-demand computation, true multitasking and have a huge internal bandwidth in their functional layer, contrasting with the sequential and limited communication bandwidth with their configuration layer, through the FPGA’s reconfiguration mechanism (i.e., ICAP\(^1\)).

\(^1\)The bandwidth delivered by Virtex-4 ICAP is in the same range as for modern hard-disk drives.
A reconfigurable application is composed of a number of logic circuits that cooperate together to implement a specific functionality. As these circuits are dynamically executed upon demand, following the terminology used in software OS, they receive the name of hardware tasks. In a similar way that a software task is loaded from hard-disk to RAM memory prior to be executed, a hardware task $\theta_i$ needs to be configured on the FPGA, i.e., its configuration bitstream must be uploaded to the configuration layer. Furthermore, as system resources are allocated to a software task, e.g., CPU time and memory, they are also allocated to a hardware task $\theta_i$, e.g., FPGA resources contained within a rectangular region of size $h_{x,i} \times h_{y,i}$ (see Figure 3.5).

Hence, hardware tasks comprise two different phases: (1) a set-up phase of duration $t_{ICAP,i}$, that is proportional to the size of the task, and (2) an execution phase of duration $t_{E,i}$, that depends on the complexity of the computation it performs, implemented data-parallelism and used clock frequency. Consequently, two different deadlines are defined for real-time hardware tasks: (1) an execution (absolute) deadline, $D_i$, and derived from this, (2) a set-up (absolute) deadline, $D_i^* = D_i - t_{E,i}$ [Dittmann and Frank, 2007]. We will not go into further details on this at this stage as the particular computing scenario defined by us is thoroughly described in chapter 4. Table 3.1 and 3.2 highlights the major differences between software and hardware tasks during compile-time and runtime, respectively.

FPGAs open the door for more sophisticated and powerful computing systems in which the available resources can be dynamically utilised in order to efficiently execute several concurrent hardware tasks. However, the fact that there is a single reconfiguration port (i.e., ICAP) in an FPGA introduces a configuration bottleneck that restricts the intensive exploitation of hardware parallelism. $U_{ICAP}$ refers to the utilisation of the ICAP and permits to characterise the real-time requirements of a task-set $\phi$ (see Equation 3.1).

$$U_{ICAP} = \sum_{\theta_i} \frac{t_{ICAP,i}}{D_i^*}$$  \hspace{1cm} (3.1)

As shown in Figure 3.5, when projecting on FPGA's reconfigurable area over time, a 3-D Computation Cube (CC) is created, whose volume increases with time: $CC(t) = t \cdot H_x \cdot H_y$. In this context, every hardware task $\theta_i$ defines a Computation Volume (CV) to be inserted in the CC: $CV(\theta_i) = (t_{ICAP,i} + t_{E,i}) \cdot h_{x,i} \cdot h_{y,i}$. Note that inserted tasks in the CC must not over-
### Table 3.1: Comparison of hardware and software (mono-core) task compile-time characteristics (adapted from [Walder, 2005])

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Software Task</th>
<th>Hardware Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design</strong></td>
<td>High-level programming languages (e.g., C, C++) or assembler.</td>
<td>Hardware description languages (e.g., VHDL, Verilog) and more recently high-level languages (e.g., SystemC).</td>
</tr>
<tr>
<td><strong>Execution Engine</strong></td>
<td>CPU</td>
<td>FPGA</td>
</tr>
<tr>
<td><strong>Size</strong></td>
<td>Measured in number of bytes.</td>
<td>Measured in number of FPGA resources (e.g., CLBs, DSP48s, BRAMs).</td>
</tr>
<tr>
<td><strong>Building Blocks</strong></td>
<td>Consists of a number of instructions to be sequentially executed.</td>
<td>Consists of a number of hardware resources which form a logic circuit.</td>
</tr>
<tr>
<td><strong>Starting and End Point</strong></td>
<td>Defined entry point. Can have several end points. Activated by setting the PC to the address of the entry point.</td>
<td>Implements a control wire to reset the circuit. Activated by resetting the circuit.</td>
</tr>
<tr>
<td><strong>Execution Structure</strong></td>
<td>Sequential (no real multitasking, no real parallelism within task, computing in time). Can call subroutines.</td>
<td>Several parallel areas of activity (real multitasking, computing in space). No subroutine calls.</td>
</tr>
<tr>
<td><strong>Execution Speed</strong></td>
<td>Defined by frequency of the clock. Maximal clock frequency defined by CPU.</td>
<td>Defined by frequency of the clock and parallelism. Maximal clock frequency defined by the longest critical path.</td>
</tr>
<tr>
<td><strong>Context</strong></td>
<td>Context is represented by content of CPU registers and stack.</td>
<td>Context is represented by all storage elements within the task (e.g., BRAMs, flip-flops).</td>
</tr>
<tr>
<td><strong>OS Interaction</strong></td>
<td>By OS kernel calls (subroutine calls): Data exchanged via memory positions.</td>
<td>OS service: Data exchanged via hardware ports (data and control wires).</td>
</tr>
</tbody>
</table>
Chapter 3- State of the Art of Operating System Support for FPGAs

<table>
<thead>
<tr>
<th>Function</th>
<th>Software Task</th>
<th>Hardware Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activation</td>
<td>Jump to entry point.</td>
<td>Load bitstream, reset and activate clock signal.</td>
</tr>
<tr>
<td>Abort</td>
<td>Interrupt and remove task from task list.</td>
<td>Stop clock and mark the assigned resources as free.</td>
</tr>
<tr>
<td>Preempt</td>
<td>Interrupt and save context registers and stack.</td>
<td>Stop clock, read-back and extract context information.</td>
</tr>
<tr>
<td>Restore</td>
<td>Restore context registers and stack.</td>
<td>Insert context information into raw-bitstream, load bitstream and activate task.</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison of hardware and software task runtime functions (adapted from [Walder, 2005])

lap in any dimension (i.e., area and time). The utilisation of the computational resources of the FPGA at a given time, $U_{COMP}(t)$, is defined as the proportion of the CC filled by all of the tasks executed up to then or being executed in that time. Note that this term depends on the triggering of hardware tasks and thus varies over time. A time-independent expression which characterises the computation requirements of a task-set $\phi$ can be deduced assuming periodic hardware tasks with the period equal to their execution deadline. In the latter situation, and when the observation time is equal to the hyper-period $t = T_{HP} = GCD(D_i)$ (GCD stands for the Greatest Common Divisor), $T_{HP}/D_i$ instances of each task are completely executed and thus, $U_{COMP}$ can be expressed as given in Equation 3.2.

$$U_{COMP} = U_{COMP}(T_{HP}) = \frac{1}{H_x \cdot H_y} \sum_{\forall i} \left( t_{ICAP,i} + t_{E,i} \right) \cdot h_{x,i} \cdot h_{y,i}$$

A feasible schedule can only be produced when both $U_{ICAP}$ and $U_{COMP}$ remain equal or less than the unit. This is a necessary but not sufficient condition to guarantee the schedulability of a task-set.

Filling the CC with hardware tasks in order to best exploit the computing power delivered by the FPGA is considered to be NP-hard\(^2\) [Steiger et al., 2004]. Efficient scheduling and allocation algorithms are required to deal with the most important enemy to beat in

\(^2\)This problem is comparable to the two-dimensional strip packing NP-hard problems, where a set of two dimensional boxes (area domain) are to be packaged within the minimal vertical strip (time domain) [Lodi et al., 2002].
Chapter 3- State of the Art of Operating System Support for FPGAs

Figure 3.5: Hardware task, Computation Volume (CV) and Computation Cube (CC)

the 3-D computation space: fragmentation (see Figure 3.6). External fragmentation is pro-
voked due to the continuous execution of hardware tasks with different shapes, sizes and
execution times. When the CC is fragmented, it can happen that a hardware task cannot be
inserted although the sum of all scattered fragments is greater than the CV of that task. On
the other hand, internal fragmentation appears when allocating more area than required by
a task. This typically occurs in traditional slotted systems, where the slots are enlarged to
the biggest task they are to allocate.

The effect of fragmentation is exacerbated by two technological features of current FP-
GAs: (1) resource heterogeneity and (2) sequential nature of ICAP. While non-homogeneous
reconfigurable areas of modern FPGAs constrain the allocatability of hardware tasks based
on the type and arrangement of resources they use, the serialised access to the reconfigu-
ration mechanism reduces the exploitable computation volume within the CC. Since only
one task can be configured on FPGA at any time, the resources that are not assigned to any
hardware task are wasted for the whole set-up time.
An important aspect that is being neglected in this analysis for simplicity is how to make hardware tasks accessible to exchange control and data information with them. Note that this issue has been studied in chapter 2.

In practice, fragmentation, ICAP exclusiveness, and sub-optimal allocation decisions lead to unfeasible schedules for task-sets meeting the above-presented feasibility conditions (i.e., $U_{ICAP} \leq 1$ and $U_{COMP} \leq 1$). Nonetheless, $U_{ICAP}$ and $U_{COMP}$ are still useful to characterise the timing and resource requirements of a given task-set. In fact, similar metrics have been proposed by other authors with the same objective (e.g., [Danne and Platzner, 2005, Walder, 2005]).

### 3.3 Reconfigurable Operating System (ROS)

The term ROS was coined by Dr. Brebner [Brebner, 1996] and it is essentially referred to a software OS augmented with functions to manage reconfigurable hardware and execute hardware applications on it. The rational of the ROS is to hide complexity by offering a set of useful services to the application developer. Indeed, arguing that a ROS deals with fragmentation of an FPGA’s reconfigurable surface and relocation of task bitstreams on behalf of the user, some authors have traced links between virtual memory in an OS and virtual hardware in a ROS [Brebner, 1996, Wigley and Kearney, 2001] (see Figure 3.7). The services offered by the ROS should be accessible through an Application Programming Interface (API) and should provide runtime support for both task management and FPGA resource management. In [Wigley and Kearney, 2002], the fundamental services to be implemented in a ROS are identified: task loading, memory management, scheduling and allocation, communications (both hardware-hardware and hardware-software) and input/output. More recently, task preemption service has also been pointed as necessary [Lubbers, 2010].
The ROS should be only aware of the sequence, dependency, resource and timing constraints of the tasks. It should not be aware of the low-level implementation details nor the function they implement. In the opposite direction, the tasks should not be aware of how the services offered by the ROS are implemented, they simply should access these services through a set of well-defined interfaces.

Various attempts to build a ROS for FPGAs can be found in the technical literature. The most significant ones are summarised in the following paragraphs.

OS4RS was a very early ROS prototype developed by IMEC with the main focus of giving runtime support for multimedia applications [Mignolet et al., 2003]. Unfortunately, very little information is provided about OS4RS implementation and functioning. Most of the information is related to the major innovation proposed: the possibility to interrupt a hardware task and restart it in software, or vice versa. Notably, this idea has inspired later work (e.g., [Zhou et al., 2005, Pellizzoni and Caccamo, 2006]). Towards this end, the same authors presented a design environment, called OCAPI-xl, to develop hardware and software versions of the same tasks with equivalent internal state representations.

In [Blodget et al., 2003], Xilinx Inc. provided the XPART API, which was intended to ease the management of FPGA resources. Unfortunately, XPART was rapidly discontinued. In [Williams and Bergmann, 2004], the authors created a Linux driver for the ICAP, and used it in an embedded Linux distribution, namely uClinux\(^3\), running on a Xilinx MicroBlaze processor. This can be considered the first successful attempt to make reconfigurable hardware

\(^3\)http://www.uclinux.org
easily accessible by a software-centric programmer, who indeed could use the ICAP from a shell script. Later, the same authors completed their work with a Linux driver that allowed FIFO-based data communications with reconfigurable hardware modules [Williams et al., 2005]. A similar ICAP driver is presented in [Donato et al., 2005]. This driver has been used to develop a Linux-based ROS with capability to manage hardware tasks as standard devices located in the /dev/ directory [Santambrogio et al., 2008]. Finally, another Linux-based ROS is reported in [Kosciuszkiewicz et al., 2007], where hardware tasks are executed on Xilinx PicoBlaze processors and communicate with software tasks using FIFO buffers. We note that the software implementation of this ROS may potentially lead to significant time overheads when the system workload is high.

HybridThreads (HThreads) has been developed by the University of Kansas [Andrews et al., 2005]. HThreads allows programmers to run software and hardware threads, simultaneously on a CPU and on an FPGA. Notably, scheduling, communication and synchronisation services are implemented in hardware, bringing significant performance benefits, and abstracted by means of a uniform POSIX-like API. However, in HThreads the hardware threads remain allocated on the FPGA even when they are idle, i.e., reconfiguration is not used. Consequently, HThreads cannot be considered a complete ROS as it fails to manage FPGA resources, i.e., FPGA resources are not shared among the threads.

BORPH was developed by the University of California, Berkeley [So, 2007]. It is distributed among five Virtex-II Pro FPGAs: one of them acts as master (control FPGA) and the remaining four implement some control logic (called uK) and allocate the hardware tasks. Because of this, these FPGAs are named user FPGAs. The control FPGA is connected to the SelectMAP pins of the user FPGAs through a point-to-point, bi-directional, 8-bit bus running at 50 MHz. This bus serves the dual role of configuring the hardware tasks in the user FPGAs, and communicating with the uKs after the functions are configured. BORPH offers a UNIX-like API. Its software kernel is an extended version of Linux 2.4.30 [So and Brodersen, 2008], which runs in a PowerPC 405 core in the control FPGA. Communication between hardware and software tasks is implemented by FIFOs and mapped to file descriptors. In BORPH, hardware tasks are assigned to user FPGAs in one-to-one fashion, leading to a very inefficient exploitation of hardware resources. Furthermore, the amount of concurrent tasks
running on the system is limited by the number of user FPGAs. In this context, BORPH does not require any specific scheduling or allocation algorithms.

ReconOS was developed by the University of Paderborn and can be seen as a porting of BORPH to a single FPGA, making special emphasis on real-time performance [Lubbers, 2010]. The user FPGAs of BORPH are assigned separate reconfigurable slots in the same FPGA in ReconOS. These slots are coupled with a control logic (called OSIF), which implements the same function as uK does in BORPH. Being contained in a single FPGA, the user functions are configured through ICAP, and communications are performed through an on-chip bus running at 100 MHz. In light of achieving real-time performance ReconOS offers an eCOS-based API, which is extended with specific system calls to manage hardware tasks. Allocation and scheduling do not deserve special attention in ReconOS: scheduling decisions are made by the eCOS kernel and allocation decisions are trivial as there are only a few slots where to map the hardware tasks.

FOSFOR was developed by the University of Nice Sophia, University of Rennes and Thales Group and is very similar to ReconOS. The most significant differences to be noted are the use of an RTEMS-based API and a NoC to interconnect the reconfigurable slots [Müller et al., 2005].

Another slotted architecture designed to serve as a platform to develop a ROS was developed at ETH Zurich [Walder, 2005]. It comprises several reconfigurable slots which span the whole height of a Virtex-II FPGA. The slots are coupled with some control and communication logic, which is located in their top part. The major innovation of this architecture is that various slots can be combined together to allocate tasks of different widths. Despite the fact that this system is provided with scheduling and allocation services, as well as FIFO-based communications, it does not include any abstracting API and therefore, it cannot be considered a ROS in its broadest sense.

A recent approach that is conceptually close to ReconOS is FUSE, developed by the Simon Fraser University [Ismail and Shannon, 2011]. FUSE relies on a slotted reconfigurable system which is implemented on a Virtex-5 FPGA and provides an embedded Linux-based API with POSIX threads running on a MicroBlaze core. Two features of FUSE are especially interesting. First, shared memories are used to exchange data between the software and

\[\text{http://ecos.sourceforge.org}\]
Chapter 3- State of the Art of Operating System Support for FPGAs

hardware tasks, thus reducing data communication overheads. Second, each hardware task is associated a Loadable Kernel Module (LKM) that implements miscellaneous device driver functionality, allowing to treat hardware tasks as memory-mapped I/O device peripherals.

Finally, CAP-OS is being developed by Karlsruhe Institute of Technology and it is intended to handle a variety of processors and accelerators under real-time constraints, using Virtex-4 FPGAs [Gohringer et al., 2010]. The API offered by CAP-OS is based on Message Passing Interface (MPI)\(^5\). As proposed in OS4RS, the computations in CAP-OS are to be performed either in software (i.e., by any of the processors), or in hardware (i.e., as a co-processor). However, the currently presented prototype only supports executing software tasks, which can be loaded into the processor's program memory either through a Network-on-Chip (NoC) or through the ICAP. Notably, CAP-OS uses a priority-based scheduling algorithm which considers both ICAP exclusiveness and inter-task dependencies. The latter scheduling algorithm as well as other CAP-OS processes run as separate threads in an embedded processor (e.g., MicroBlaze or PowerPC 405), which is equipped with Xilinx Xilkernel multithreading solution. In the future, the authors expect to include the capability to configure hardware tasks upon request by the processors as well as to modify the number of processors in the system. Towards this end, bitstream relocation is pointed as necessary, leading us to understand that the current CAP-OS prototype relies on a slotted architecture.

3.3.1 (Real-Time) Scheduling of Hardware Tasks

So far we have seen that scheduling and allocating hardware tasks on a partially reconfigurable FPGA is an NP-hard problem that extends in both area an time domains. Some authors decompose this problem in the two domains, proposing separate scheduling and allocation algorithms that cooperate together, while others face the whole 3-D problem, analysing both area and time domains at the same time.

In [Danne and Platzner, 2005], the authors propose two preemptive scheduling algorithms for periodic tasks: EDF Next Fit (EDF-NF) and Merge-Server Distributed Load (MSDL). In EDF-NF, the task with the closest deadline that fits in the FPGA is scheduled first, while in MSDL, tasks are successively merged into servers, which are then scheduled sequentially using EDF.

\(^5\)MPI is a language-independent communications protocol used to program parallel computers.
In [Steiger et al., 2003], the authors propose two non-preemptive scheduling algorithms for sporadic hardware tasks: horizon and stuffing. These algorithms keep track of future releases of area when the executing tasks finish and, upon a new task arrival, they simulate the future state of the FPGA to check whether there will be sufficient adjacent free area to allocate the task before its deadline expires. If not, the task is rejected. In [Chen and Hsiung, 2005], the authors propose a remedy to solve a limitation detected in stuffing, namely it always allocates tasks on the leftmost edge of the free area, achieving better results. In classified stuffing, the tasks with a high ratio between area and execution time are placed starting from the leftmost edge of the free area, while the tasks with a low ratio are allocated on the opposite way. Even better results are reported when applying stuffing over a time-window [Zhou et al., 2006]. Based on this, the same authors propose the Compact Reservation (CR) algorithm which is aimed at reducing the complexity of window-based stuffing [Zhou et al., 2007]. A similar approach is presented in [Cui et al., 2007], where the authors propose a non-preemptive algorithm, called one level look ahead, which delays the allocation of hardware tasks with the objective of reducing the fragmentation on the device.

All of the algorithms presented above neglect the reconfiguration port exclusiveness and latency of current FPGAs, i.e., they assume that reconfiguration does not take time, and therefore, they are not suitable to be implemented using real hardware. The next generation of scheduling algorithms do consider this restriction.

In [Dittmann and Frank, 2007], the authors propose to schedule access to the reconfiguration port of the FPGA based on the allocation deadlines of the tasks. They port traditional real-time scheduling algorithms for mono-core processors to reconfigurable hardware (i.e., preemptive EDF and DM).

In [Lu et al., 2009b], the authors propose to schedule the reconfiguration port access with the objective of optimising the FPGA utilisation, but they do not consider any real-time constraints for the tasks. An interesting idea proposed in this work is the possibility of reusing the already configured tasks on the device, even when they implement only a part of the total functionality required: computation can start in these tasks, while the task that implements the complete functionality is being configured, and once the latter is ready, the partially processed data can be transferred to it. In [Marconi et al., 2010], the same authors propose the so-called 3D Total Contiguous Surface (3DTCS) heuristic that is intended to
avoid task placements that will be an obstacle for other incoming tasks in the future. 3DCTS computes the total contiguous surface of the CV defined by a given task with the CVs of other tasks in the CC and with the CC boundaries. Therefore, higher 3DTC value will result in more compaction in space and time.

A third generation of more advanced scheduling algorithms, which are aware of task-dependencies and data communications, is currently being developed. An example can be found in [Lu et al., 2010], where the authors specifically consider data communications between the hardware tasks and external devices. In [Gohringer et al., 2010], the scheduling decisions are made using static priorities that are assigned based on the amount of communicating tasks. Finally, in [Redaelli et al., 2009], a reconfiguration-aware heuristic scheduler is proposed, which is aimed at exploiting configuration prefetching, task reuse, and anti-fragmentation techniques.

### 3.3.2 Allocation of Hardware Tasks

Most of the research efforts carried out up to date in the task allocation field assume a 2-Dimensional area model and consider the tasks to be relocatable rectangles that can be placed anywhere on the FPGA device.

The pioneering work described in [Bazargan et al., 2000] propose to Keep track of All the Maximal Empty Rectangles (KAMER) or only of the Non-overlapping Empty Rectangles (KNER) in the device. Note that tasks can be allocated on these empty rectangles without overlapping other tasks already allocated. When allocating a new task in an empty rectangle, some area fitting heuristics (e.g., Best-Fit, First-Fit) are used to decide whether the rectangle has to be split vertically or horizontally.

In [Walder et al., 2003], the authors propose to use a hash matrix in which every entry consists of a pointer to a list of the MERs of the corresponding size. Again, area fitting heuristics are used to select in which MER to allocate a task. In light of achieving better performance, they propose to update the hash matrix while the task is allocated.

In [Ahmadinia et al., 2004a], the authors propose to keep track of the occupied area instead of the free area, arguing that the amount of empty rectangles grows much faster than the number of occupied rectangles. Besides, they explain how to simplify the allocation problem by shrinking the area of the chip and simultaneously blowing up the placed
tasks by half the width and half the height of the task to be allocated. In this work, the Nearest Possible Position (NPP) algorithm is also presented, which is aimed at minimising the routing cost between the tasks that communicate with each other. The latter routing cost is computed based on the Euclidean distance.

In [Handa and Vemuri, 2004], the authors aimed at constructing staircases with the empty area, and finally, use these structures for finding the MERs. Likewise, in [Cui et al., 2007], the authors propose the Scan Line Algorithm (SLA) for finding MERs.

In [Morandi et al., 2008] a binary tree is proposed, in which each node represents an occupied location of the device and each leaf represents a MER. This means that, when looking for a suitable location to allocate a task, only leaves have to be explored. Moreover, the authors introduce the Routing Aware Linear Placer (RALP) algorithm, which is aimed at allocating the tasks that communicate together on the empty rectangles with the shortest Manhattan distance to minimise routing costs. In [Tomono et al., 2004], the authors propose to use the boundaries of already placed tasks as routing channels.

A completely different approach is described in [Ahmadinia et al., 2007] and [Tabero et al., 2004]. In these works the authors propose to manage the empty area perimeter instead of MERs. In [Tabero et al., 2004], a Vertex List Set (VLS) is used to keep the contour information of each free area fragment in the FPGA. In this context, the authors propose the 2-Dimensional Adjacency (2DA) heuristic, whose objective is to allocate the tasks in the positions with the highest contact perimeter with other running tasks or with the FPGA boundaries. The authors state that these positions are the vertices of already running tasks, which are indeed included in the VLS.

The above idea is further developed in [Tabero et al., 2006], where the 3-Dimensional Adjacency (3DA) heuristic is proposed. The objective of 3DA is to allocate the tasks in the positions with the highest contact surfaces which are formed when prolonging in time the aforementioned contact perimeter; i.e., the execution time of the running tasks is also considered. As a result of including the time domain in the analysis, 3DA outperformed 2DA. In fact, 3DA is currently one of the most effective heuristics for efficiently allocating hardware tasks onto reconfigurable devices, being used or serving as inspiration for other approaches in the field (e.g., 3DCTS [Marconi et al., 2010]).
In [Tabero et al., 2008], the authors propose a heuristic to evaluate the fragmentation on the device, based on the amount of existing free area fragments and their shape. This heuristic is used to select the best allocation for incoming tasks in order to minimise the overall fragmentation on the device.

In [Lu et al., 2009a], the Multi-Objective Hardware Placement (MOHP) algorithm is presented, which combines some of the previously proposed ideas. In order to allocate the tasks with close deadlines and no communication requirements the First-Fit heuristic is used, i.e., tasks are allocated on the first found location where they fit. Tasks with slack deadlines and no communication requirements are allocated according to [Tabero et al., 2004, Tabero et al., 2006], and tasks that need to communicate with other tasks are allocated according to [Ahmadinia et al., 2004a].

Finally, a number of authors propose to compact the allocated tasks on the FPGA from time to time in a similar way that the hard-disk of a computer is defragmented [Li and Hauck, 2002, Ejnioui and DeMara, 2005, van der Veen et al., 2005, Kalte and Porrmann, 2005, El Farag et al., 2007]. However, defragmentation techniques incur high reconfiguration overhead provoked by extra task relocations.

We note that most of the approaches herein described use a very abstract device model, which indeed can be considered incomplete as it does not account for some physical constraints of FPGAs (e.g., granularity of reconfiguration). References [Ahmadinia et al., 2004c] and [Montone et al., 2008] are some of the only works which consider these issues. In the former work, the authors propose the Least-Interference Fit (LIF) heuristic with the criteria of interfering running tasks as least as possible when allocating new incoming tasks. However, they exclusively address column-based reconfigurable FPGAs (e.g., Virtex-II). In the latter work the authors target modern tile-based reconfigurable devices (e.g., Virtex-4).

### 3.3.3 Switching Hardware Tasks

While the context information of a software task is clearly defined, i.e., program counter (PC), registers, and stack, the context of a hardware task is not so clear and uniform, including the state of all of the synchronous components of the task, i.e., memories and registers (see Table 3.1). Unfortunately, the latter is usually vast and depends on the implementation of the task.
Two techniques have been proposed up to date for switching hardware tasks in partially reconfigurable FPGAs. The first method consists in providing task's registers and memories with extra interfaces to enable access when saving and restoring task's context (e.g., [Ahmadinia et al., 2004b, Kalte and Porrmann, 2005, Jovanovic et al., 2007, Koch et al., 2007]). These could be implemented as scan chains, memory-mapping structures, or scan chains with shadow registers [Tuan and Amano, 2008]. The main advantage of this intrusive method is data efficiency as only the required information is saved. On the other hand, its major problems are the area overhead introduced by the added interface logic, which usually reduces the maximum usable clock frequency as well, and the difficulty for designing standard generic interfaces for different type of tasks. The second method consists in harnessing the reconfiguration port of the FPGA for reading-back and later restoring the context information in the bitstream domain (e.g., [Simmler et al., 2000, Ahmadinia et al., 2004b, Kalte and Porrmann, 2005, Jozwik et al., 2010]). While transparency is the key benefit of this method, it incurs significant time overheads due to the limited access speed through the reconfiguration port and the necessity of accessing complete configuration frames, which include context information together with other useless configuration data. Furthermore, technological limitations also exist due to the difficulties for individually accessing registers when resuming their state.

In [Ahmadinia et al., 2004b], the authors propose to compress the read-back bitstream to reduce the memory requirements, while in [Tuan and Amano, 2008], the authors propose a method for identifying the instants when the context information is relatively small to be selected as preemption points (e.g., between the processing of two data blocks).

In [Koch et al., 2007], the authors use checkpoint-rollback recovery with hardware tasks. This is a commonly used technique in the software field which consists in taking checkpoints (i.e., error-free contexts) and using them after an error occurred to re-establish a consistent state.

3.3.4 Inter-Task Communications and Synchronisation

The existing solutions for providing inter-connectivity to relocatable hardware tasks have already been described in chapter 2. Adaptable on-chip buses, NoCs and online routing have been proposed towards this end. However, most of currently available ROS rely on
a static communication infrastructure which interconnects all of the reconfigurable slots in the system, where the hardware tasks are allocated, and the main processor, where the software part of the ROS runs (e.g., [Walder, 2005, Lubbers, 2010, Gohringer et al., 2010]). In order to communicate between software and hardware tasks, most ROS resort to wrap the hardware tasks with software wrappers and then use existing mechanisms in standard software OS (e.g., Linux [Williams et al., 2005, Bergmann et al., 2006], eCOS [Lubbers, 2010] or RTEMS [Muller et al., 2005]).

As with inter-task communication, most of the currently proposed solutions for synchronising hardware tasks rely on static signals (e.g., [Lubbers, 2010]).

However, as presented in chapter 2, the existence of static wires crossing the FPGA surface involves important limitations in terms of relocatability of partial bitstreams. Indeed, fully relocatable hardware tasks with support for inter-communications and synchronisation currently belong only to theory. We note that this is one of the major divergences between research efforts that study the RC paradigm from an algorithmic perspective, e.g., work reported in section 3.3.2, and the efforts that are aimed at building real prototypic solution, e.g., works reported in section 3.3. A secondary problem is that the static communication and synchronisation infrastructure occupies a significant amount of on-chip resources and usually leads to low usable clock frequencies. For instance, the NoC reported in [Bobda et al., 2005a] consumes nearly all the resources of the FPGA chip, and the bus clock frequency reported in [Sedcole et al., 2007] is only 50 MHz.

### 3.4 The Next Abstraction Horizon: Hypervisors vs. ROSs

While an OS simply abstracts machine’s hardware to the user applications, an hypervisor provides multiple isolated Virtual Machines (VMs), each executing its own OS and user applications [Popek and Goldberg, 1974]. In order to create the illusion that there are multiple “virtual” instances of a limited number of physical computer resources (e.g., CPU registers, interruptions, timers, IOs and MMU), the hypervisor takes control over the system calls that manipulate those resources. This technique is known as para-virtualisation and requires small changes to be made in the guest OSs, namely to redirect the system calls to the hypervisor (i.e., hypercalls). Besides, the hypervisor schedules the execution of the different VMs in the virtualised platform and provides communication support for them. In order
to allow the use of guest RTOSs in the VMs, fixed priority-based hard-real time scheduling algorithms are typically used (e.g., XtratuM [Crespo et al., 2009]).

The VMs are allocated in separate address spaces and run securely and independently from each other (i.e., a VM cannot access the virtual resources assigned to other VMs). This is very useful in the scope of mixed-criticality systems (e.g., safety-critical and noncritical OSs and applications can run in different VMs), multi-level security systems (e.g., each VM can be assigned a different security level), software migration (e.g., legacy OSs and applications can run in a VM while new applications execute in other VMs) and multi-cores (e.g., applications that were originally created for mono-core processors can take advantage from a multi-core platform). On the other hand, the main drawback of hypervisors is the increased overheads due to the extra software abstraction layer that is added.

Both hypervisors and ROSs allow for sharing a limited number of physical computer resources and provide the illusion that there are multiple instances of them. The difference is that a ROS virtualises low-level FPGA resources, such as BRAM, DSP48s and CLBs, which indeed can be used to implement the higher-level resources a hypervisor deals with. For instance, CLBs can be used to implement the CPU hardware (e.g., registers, timers and MMU) while BRAMs can be used to host guest OSs and user applications executable code. Notably, since “virtualisation” is implemented at a low hardware level in a ROS, there is no need to modify the guest OSs nor to insert any extra software layer. Hence, when using a ROS, VMs approach real physical machines.

### 3.5 Chapter Conclusion

This chapter introduced the most important concepts related to RC as well as the most significant research efforts to build a ROS. We noted that most of these works do not go beyond theory and simulations, and the handful of real implementations presented up to date are relatively basic and make limited use of the possibilities offered by modern partially reconfigurable Xilinx FPGAs, e.g., most of the developed ROS rely on slotted reconfigurable systems and use currently deprecated column-based reconfigurable FPGAs. The next chapters are aimed at describing our own attempt to design and implement a novel ROS, which we believe will help to promote the exploitation of most of capabilities delivered by currently available FPGAs, e.g., slotless reconfiguration and partial bitstream relocation.
Overview of R3TOS and Associated Computing and Dependability Models

Our Reliable Reconfigurable Real-Time Operating System (R3TOS) deals with all the typical issues of RC described in the previous chapter. R3TOS exploits the reconfiguration capabilities offered by modern Xilinx FPGAs in an innovative way to satisfy the often conflicting requirements of real-time, fault-tolerance and high-level programming.

R3TOS bridges the gap between high-level software and low-level FPGA hardware by offering OS-like kernel support for reconfigurable applications. It allocates FPGA resources (e.g., DSP48s, BRAMs, CLBs) to the hardware tasks as if they were traditional processor resources (e.g., CPU time, memory), but, unlike in traditional processors, R3TOS delivers real hardware multitasking and logical correct computation despite the occurrence of faults. Indeed, it is capable of detecting, isolating and circumventing any damaged resource on-chip.

Moreover, in order to guarantee a good quality of service, R3TOS schedules its workload based on real-time criteria. Despite the novel ideas of R3TOS, its objective is not to break away from traditional software-based computation, but to combine it with hardware-based computation to build powerful reconfigurable applications, where “everything” is adaptable to the computing demands.

This chapter serves as an introduction to R3TOS, highlighting its main capabilities and features, prior to analysing each of its parts in subsequent chapters.

Most of this chapter has been accepted for publication in [Iturbe et al., 2013b].
4.1 R3TOS Foundations

R3TOS is founded on the basis of resource reusability and computation ephemerality. It makes intensive use of reconfiguration at the finest granularity of Xilinx FPGAs, keeping the resources available to be used by any incoming task at any time. Hence, computing tasks and associated supporting circuitry (e.g., inter-task communication channels, clock distribution wires, etc.) are configured when required, and removed when they are no longer needed. Therefore, FPGA resources can be used either for computation or for communication purposes at different times. Indeed, R3TOS does not rely on any pre-routed communication infrastructure, instead it creates on-demand communication channels among the tasks on-the-fly.

In R3TOS, the control logic to drive the tasks is attached to their own circuitry, making them self-contained and closed structures which are fully relocatable within the FPGA. Hence, the tasks only include the necessary amount of resources to perform their computation and therefore, internal fragmentation is reduced considerably. This is a completely different approach when compared to related state of the art, where (enlarged) hardware tasks are executed in pre-defined reconfigurable slots coupled with fixed control logic and connected to a static communication infrastructure to exchange data among them.

The Task Control Logic (TCL) includes an Input Data Buffer (IDB), an Output Data Buffer (ODB), and a Hardware Semaphore (HWS) to enable/disable computation. TCLs provide a means to virtually lock physical data and control inputs/outputs of the hardware tasks to logical positions in the configuration memory of the FPGA. Since the TCLs are accessible through the configuration interface whichever memory positions they are mapped to, the allocatability of the tasks is not constrained by the position of the communication interfaces decided at design time anymore. Furthermore, this scheme improves multitasking capabilities as the number of tasks that can be concurrently executed on FPGA is not limited by the amount of communication interfaces defined at design time. Finally, note that the fact of replacing “physical” Bus Macros (BMs) with “logical” TCLs is in consonance with the current trend to hide hardware related aspects to the designer. E.g., the latest release of Xilinx tools to develop reconfigurable systems are able to manage the BMs on behalf of the designer, who has not to worry about where and how many BMs are instantiated.
There are several immediate advantages resulting from the innovative use of FPGA resources in R3TOS:

- The allocatability of the tasks is improved as the FPGA area is kept free of non-necessary obstacles at all times; e.g., static routes. This results in higher flexibility to allocate the tasks around the damaged resources (improved fault-tolerance) and to increase the computation density by compacting the tasks in the chip (improved efficiency). Furthermore, the complexity of the allocation algorithms is simplified as they do not need to be aware of any underlying implementation-related irregularities in the reconfigurable area. Note that a traditional reconfigurable system must preserve the static routes, resulting in additional difficulties which penalise the performance.

- Tasks can be deallocated very quickly using MFWR commands as there is no need to preserve any element within the region occupied by them, i.e., task deallocation simply consists in blanking the whole content of the corresponding configuration frames.

- Dynamic power consumption is substantially reduced as only the strictly necessary resources on the FPGA device are active at any time [Zhang et al., 2006].

- The fact that we separately and individually feed each task with its highest allowed clock frequency clearly outperforms the easy and more usually chosen option of clocking the entire system at the slowest rate [Iturbe et al., 2012]. Note that this is consonance with the objective of semiconductor manufacturers of creating circuits able to work at ever higher frequencies and with the objective of synthesis tools developers of achieving better circuit implementations with shorter critical paths. Moreover, if required, the clock frequency can be chosen with the objective not to increase performance, but to reduce the power consumption and heat dissipation.

- Since R3TOS relocates the circuitry along the entire device, switching activity naturally tends to distribute among all the resources. As a result, the wear in the device is leveraged, delaying the occurrence of damage [Srinivasan et al., 2006, Feng et al., 2010, Angermeier et al., 2011]. This does not happen in traditional systems where some of the resources are prone to fail earlier due to intensive use, e.g., the resources used to implement the static communication infrastructure.
Moreover, R3TOS allows for a new *computing scenario* where FPGA resources are dynamically used to build highly adaptable multi-core systems where not only the number of cores, but also their data-path and executing code can be configured at any time to exploit both data and process level parallelism (see Figure 4.1). In [Hill and Marty, 2008], the authors name such systems as *dynamic multi-cores* and use Amdahl’s law to estimate achievable speed-up figures \(^1\).

Due to the similarity with the software hypervisor technology, R3TOS is also expected to contribute in the building of multi-level security and mixed-criticality systems.

Finally, despite it is not directly addressed in this thesis, R3TOS is able to configure multiple identical task instances (i.e., *cloned tasks*) using MFWR commands with very small time overheads [Ebrahim et al., 2013]. This allows to approach the building of dynamically customisable Single Instruction Multiple Data (SIMD) computers with capability to exploit specialised data level parallelism without suffering from the von Neumann bottleneck; i.e., the *cloned* tasks are to perform the same processing on multiple data points simultaneously.

On the other hand, the limitations associated to R3TOS mainly come from the reconfiguration bottleneck provoked by the ICAP. First, the configuration of a hardware task delays its execution by a non-negligible amount of time. Second, the configuration of on-demand communication channels among the tasks incurs an overhead which is significantly greater when compared to the time needed for establishing a virtual connection through a NoC or to the nearly-zero communication delay in an on-chip bus.

\(^1\)The authors of this paper have made available a tool in the web to check the achievable speed-up factor in different parallel situations [http://research.cs.wisc.edu/multifacet/amdahl](http://research.cs.wisc.edu/multifacet/amdahl)
In effect, the fully reconfigurable nature of R3TOS introduces a reconfiguration overhead which, in some applications, can be greater than that introduced by traditional reconfigurable systems. Note that a similar effect can be seen in a traditional software OS, whose background processes, that increase the underlying machine's flexibility and make it easy-to-use also hinder the achievable performance brought about by directly working on the machine's bare hardware. Nonetheless, the gained flexibility is absolutely necessary to achieve the high fault-tolerance and adaptivity levels accomplished by R3TOS.

It is true that the performance improvement due to more flexible use of FPGA resources is constrained by the also more reduced communication bandwidth resulting from the lack of any pre-routed communication infrastructure in the system. Note that this fact is not directly considered by the aforementioned Amdahl's law [Hill and Marty, 2008]. But, it is also true that R3TOS approaches the entropy of the configuration information, resulting in faster reconfiguration of the tasks in most of the cases. As shown in Figure 4.2, R3TOS circumvents all the unnecessary configuration data which is added to the tasks in traditional reconfigurable systems, e.g., static routes crossing the slots and extra configuration information resulting from the enlargement of the tasks to the slots. This contrasts with what occurs in slotted traditional reconfigurable systems, where configuration data is likely to be distributed among a great number of lightly-used configuration frames.

To minimise reconfiguration overheads R3TOS promotes the reuse of both previously configured tasks and intermediate partial results computed by them. Furthermore, R3TOS exploits the data movement that naturally occurs while performing the computation as a way to reduce communication requirements; i.e., data can be directly accessed from the resources where the producer tasks leave their results.

4.2 The R3TOS Computing Model

R3TOS envisions a real-time multitasking scenario, which supports the benefits brought about by reconfigurable hardware (e.g., true hardware multitasking and computation specialisation), without significantly modifying the traditional software-based programming style. Indeed, task definitions and their interactions are described using parallel software programming syntax (e.g., POSIX Pthreads), but the body of some of the tasks (hardware tasks) is implemented in hardware.
The term ‘hardware task’ is used to reflect the fact the task relies on specific purpose custom circuitry to perform computation, but this does not mean it does not include any software component inside. Indeed, the custom circuitry could be used as a hardware accelerator by a processor running a software program inside the task.

The work carried out in this thesis is mainly focussed in the management of hardware tasks. Software tasks are not directly addressed and they are assumed to be executed, as usual, on standard processors. Therefore, unless it is not explicitly indicated, the term ‘task’ must be understood as ‘hardware task’ henceforth.

R3TOS addresses two main types of hardware tasks: (1) **Data-stream processing tasks**, to be used in data-intensive applications with regular dependencies, and (2) **Hardware-acceleration tasks**, to speed-up the execution of computation-intensive portions of software code, which indeed can be translated into hardware using high-level synthesis tools (e.g., Xilinx Vivado).

As depicted in Figure 4.3, R3TOS is built upon a hybrid hardware-software abstracting execution environment.

---

2 Regular data dependencies are those which repeat throughout the set of computations.
At design time, the user can rely on the R3TOS API to program his/her reconfigurable application. The API includes a set of System Calls to give seamless support for both software and hardware task invocation (e.g., similar to POSIX `pthread_create()` or `fork()`), synchronization (e.g., similar to POSIX `pthread_join()` or `sem_wait()`) and inter-task communications (e.g., equivalent to POSIX `msgsnd()` and `msgrcv()`). Therefore, the user only needs to describe his/her reconfigurable application; e.g., triggering conditions for each task, real-time requirements, missed deadlines handling, etc.

At runtime, the execution of the tasks is controlled by a main CPU based on the user specifications. Basically, the main CPU executes a program which is conceptually similar to a traditional RTOS. Indeed, it is based on a software real-time microkernel (SWuK), which is extended with extra functionality to interact with the hardware microkernel (HWuK). Hence the main CPU schedules-executes software tasks and forwards hardware tasks to HWuK.

The HWuK gives support to the main CPU to deal with hardware tasks, serving as the substrate upon which the hardware-related services offered by the R3TOS API are built. Namely, R3TOS HWuK includes: (1) a Scheduler server that is specifically designed for scheduling hardware tasks, (2) an Allocator server to manage FPGA resources, and (3) a Configuration Manager to translate the high-level operations dictated by scheduler and allocator servers into reconfiguration commands for the FPGA. Note that the latter implements most of the Hardware Abstraction Layer (HAL) of R3TOS (see Figure 4.3).

The configuration information to build the circuitry required by the hardware tasks, including the executable code of the software components inside, if any, is stored in a Bitstream Memory. At runtime the hardware tasks are allocated to different positions within the FPGA by appropriately changing the frame addresses when their bitstreams are transferred from the bitstream memory to the configuration memory of the FPGA. To achieve the highest reconfiguration bandwidth, the frame relocation process is done by a dedicated
hardware, operating at the highest allowed clock frequency by the ICAP. Consequently, the limited physical FPGA resources are used to implement a large virtual computing resource which is time-shared to serve multiple hardware tasks upon request. Each of the tasks uses its own private piece of FPGA resources; i.e., user-space logic and memory.

To improve performance, the configuration memory of the FPGA is used as a cache for both tasks and data. In fact, hardware tasks are deallocated from the FPGA only when their resources are required by other coming tasks, and the partial results computed by them are retrieved only when they are required by a software task.

Both software and hardware tasks are assigned a space in the Main Memory attached to the main CPU to hold local data and, in the case of software tasks, to hold the executable code as well. Since the main memory is shared between both hardware and software tasks, here is where the interactions between both types of tasks occur, as defined by the user at design time and, as implemented by the main CPU at runtime. An advantage of this scheme is its compatibility with most software compilation systems and most task memory mappings of software OS.

To simplify hardware-software communications, a fixed region within the main memory is shared between the main CPU and the HWuK to exchange data through. This region is organised in the form of an ODB and an IDB. The data written by the CPU in the ODB is delivered by the HWuK to the corresponding hardware task running on the FPGA. Likewise, the data written by the HWuK in the IDB is relocated by the CPU to the data segment assigned to the corresponding software task in the main memory. Therefore, the HWuK cannot directly access the data segments of the tasks in the main memory and the main CPU cannot access the hardware tasks in the FPGA, guaranteeing no interference between them.

This section describes the way a reconfigurable application is designed and executed in the real-time multitasking computing environment defined by R3TOS (see Figure 4.4).

### 4.2.1 Application Design Phase

The design phase starts with the conceptualisation and refinement of specifications to produce a behavioural model of the reconfigurable application. The decision about which functionality should be implemented in hardware and which in software is made based on estimations about cost and performance (e.g., required silicon area, hardware response
Figure 4.4: R3TOS overview
time, occupied program memory and software execution time). This codesign process is a challenge in its own right [Bertels, 2011], but it is out of the scope of this thesis. We assume that the reconfigurable application is successfully decomposed into a set of Software Functions (SFs), to be executed in the main CPU, and a collection of Processing Elements (PEs), to be implemented using FPGA resources. In general, PEs turn the fine-grained and generic FPGA resources into coarse-grained and specific computing machines.

There is no exact definition for PEs, they are assumed to be components which transform a set of input data into a set of output results using some specific-circuitry to speed-up this process. Note that the list of components which match this definition is large, ranging from small pure hardware modules (e.g., nodes of a systolic array), to complex cores which may include hardware accelerators and peripherals connected by means of buses (see Figure 4.5a). The main difference between the aforementioned SFs and the software code executed in the PEs is that the former is generic enough to be executed by a general-purpose CPU, while the latter posses specific features which are amenable to be accelerated by PE's custom architecture (e.g., ad-hoc instruction set [Zuluaga et al., 2012]).

When partitioning the reconfigurable application, related PEs are grouped together into hardware tasks to reduce the longest path. The latter is a key issue towards high-performance as R3TOS is able to execute each task at its maximum allowed clock frequency at runtime.

In a second phase, loosely coupled PEs in time or content must be extracted from the tasks until the desired granularity is achieved. Content coupled PEs assigned to different hardware tasks result in inter-task communications, where the amount of data to exchange depends on the coupling level. On the other hand, the PEs mapped to the same tasks include inherent RTL communications which benefit from the huge capability to move data among registers and memory elements delivered by massively parallel FPGAs. Hence, there is a trade-off between inter-task communication bandwidth requirement and task granularity, which ultimately influences the reconfiguration overheads and FPGA resource demands of the application.

As other related works in the field (e.g., [Jiang and Wang, 2007]), we distinguish between two types of tasks based on their communication requirements. High-Bandwidth Communication (HBC) tasks process a relatively high amount of data within a relatively
short amount of time (i.e., communication dominates computation), while \textit{Low-Bandwidth Communication (LBC) tasks} process a relatively reduced amount of data within a relatively long amount of time (i.e., computation dominates communication). Typically, data-stream processing tasks are HBC and hardware-acceleration tasks are LBC.

In general, the application partition is considered to be effective when the sum of the time needed to begin the calculation in the hardware tasks (i.e., due to reconfiguration and data/control exchange with the TCLs) and the calculation itself is smaller than the time needed to perform those calculations solely in software.

After having partitioned the reconfigurable application into hardware tasks, these are synthesised together with a wrapping TCL to obtain a single merged relocatable partial bitstream for each of them. Note that the tasks are separately synthesised and constrained to specific closed regions within the FPGA. In order to obtain “compressed” partial bitstreams, which use MFWR configuration commands, the Xilinx BitGen utility is used with the “-r” switch. The obtained partial bitstreams are post-processed with two objectives: (1) adapt them to the format required by R3TOS (see section 4.3.2 and section 6.6.2), and (2) remove the configuration information related to the tasks’ data buffers. Finally, the processed partial bitstreams are loaded in the bitstream memory. On their part, the SFs are compiled together with the application description and the R3TOS API to generate the application executable, which is loaded in the main memory.

The design phase concludes with the merging of application-dependent information (e.g., task dependencies, real-time deadlines, etc.) and target FPGA-dependent information (e.g., chip size and layout, FPGA Device ID, etc.) into R3TOS to create the appropriate execution environment for the reconfigurable application. This information is to be collected and formatted by a feature extractor script. Additionally, R3TOS is provided with specific implementation-dependent information (e.g., HWuK placement within the FPGA).

We would like to note that the flow for designing reconfigurable applications using R3TOS described in this section is amenable to be automated.

\textbf{4.2.2 Inter-task Communications and Synchronisation}

Without loss of generality, a reconfigurable application can be modelled as a Directed Acyclic Graph (DAG) where \textit{vertices} represent tasks and \textit{edges} represent inter-task communication.
Chapter 4 - Overview of R3TOS and Associated Computing and Dependability Models

channels (see Figure 4.5c). Vertices produce and consume data from edges, which in turn buffer the data in a FIFO (first-in, first-out) fashion. In R3TOS, buffering capability is mandatory as the tasks can be executed at different slots of time. Besides, synchronisation is necessary to coordinate data producer and consumer tasks when accessing their shared communication channels. The TCLs attached to the hardware tasks provide support for communications, synchronisation and data buffering.

Indeed, the TCL delivers data to be processed from the IDB to the PEs, and stores the subsequent results computed by the latter in the ODB. Hence, the data buffers of the TCL are functionally equivalent to the FIFO queues commonly inserted in-between data processing pipeline stages or to the local caches used in traditional processors. Indeed, when the main PE in the task is a processor, the buffers are directly mapped to its program memory. Otherwise, a glue logic adapts data from the way it is stored in the buffers to the needs of the PEs and vice versa (e.g., data rate, word-length). Therefore, TCLs provide hardware tasks with dedicated access to their data buffer. The data buffers of HBCTask!High-Bandwidth Communication (HBC) tasks are implemented using high-density storage resources (namely BRAMs), while the buffers of LBC tasks are implemented using low-density storage resources (namely LUT-RAMs).

After a task completes its computation, the computed results remain stored in its former ODB until they are required as input by another task. Temporarily buffering the partial results along the entire chip, which is a result of the flexible allocation scheme used in R3TOS, is the natural way to exploit the distributed nature of FPGA fabrics. Hence, hardware tasks leave a data trace in the user data plane of the FPGA when they finish. Data traces do not significantly constrain the allocatability of the tasks within the configuration plane. First, BRAMs can be by-passed when allocating new incoming tasks in overlapping locations as long as they are not specifically used by them. Refer to [Becker et al., 2007] for a detailed explanation on this. Second, data traces stored in LUTs can be moved to other locations in relatively short time.

Note that each task can produce data for more than one consumer tasks and thus, the data trace left by a producer task must be preserved while there is at least one active consumer in the application. In order to know when a data trace is not necessary anymore and can be removed from the FPGA, a counter is associated to each task. This counter is initially
Partitioning the reconfigurable application into hardware tasks

Hardware task-set. A task encapsulates a functionality and defines a set of ports that make up the interface to its environment.

Execution DAG

3-D execution

Figure 4.5: Designing and executing a reconfigurable application in R3TOS
set with the number of existing data consumer tasks for the task to which it is associated, and it is decremented with each access to the data trace. When the counter reaches 0, then that data trace can be removed from the FPGA. The typical way to deal with data-dependent computation, where the number of data consumer tasks for a data trace may change at runtime, is to initially set the counter to 1 and update it with the actual value when it is known.

Data is transmitted from a producer task $P$ to a consumer task $C$ by copying the content of $P$'s ODB to $C$'s IDB. In the case of software tasks, the data buffers are those mapped into the main memory which are accessible by the main CPU. Hence, data traces are relocated along the chip as required by the tasks. If possible, the ODB of the producer task $P$ is configured to be the IDB of the consumer task $C$ so that there is no need to relocate any data; i.e. data is in the position the consumer task expects to be. Otherwise, R3TOS harnesses the ICAP to establish on-demand “virtual” channels among the hardware tasks through the configuration layer (see Figure 4.6).

![Figure 4.6: Hardware-software inter-task communications using ICAP-based “virtual” channels](image)

To speed-up the relocation of high amounts of data in HBC tasks, R3TOS can configure physical routes to connect BRAMs when there are no obstacles between them; e.g., other tasks. The physical routes and the logic to drive the BRAMs while data is relocated are grouped together to form Data Relocating Tasks (DRTs). These are always configured before the computing tasks to which they give support with the objective of parallelising both the subsequent data relocation, which occurs through the functional layer of the FPGA, and the data consumer task allocation, which occurs through the configuration layer. The physical routes offer a potentially higher bandwidth than ICAP-based “virtual” channels, not only
because the data is read and written at once, but also because the usable clock frequency can be made higher than that of the ICAP.

As shown in Figure 4.6, the tasks perform computation in the functional layer of the FPGA and inter-task communications are carried out, or at least initiated, through the configuration layer. The synchronisation needed to coordinate access to data buffers from both layers is provided by the HWS included in the TCL. The HWS acts as the internal reset signal for the task; i.e., the task starts computing only when the HWS is enabled (by the HWuK) and once it completes the computation, the task itself disables its HWS. Therefore, the HWS is active only while the task is performing active computation and hence, it is also used to implement the exclusive access to FPGA resources.

R3TOS targets an event-triggered data-dependent computation model where data exchanges among tasks are carried out only prior to task execution, with the computation thereafter performed atomically. The tasks are triggered when all their input operands are ready to be processed. This functioning enables temporal isolation among hardware tasks execution, avoiding most of communication related problems in RC, such as deadlocks or race conditions. As a result, the system is predictable enough to approach real-time performance. We note that this functioning perfectly matches the way a traditional software OS works (i.e., the context of the task is loaded when granted with CPU/FPGA access).

### 4.2.3 Real-Time Hardware Task Model

From a computational point of view, the merger of a set of PEs and a TCL gives rise to a task with “software look and feel” which is unequivocally identified by means of a TaskID. Indeed, the main CPU can access the advanced computation capabilities delivered by the hardware tasks, without having to know anything about their implementation details, and regardless of their placement within the FPGA. The CPU simply writes the data to be processed to its ODB, which is mapped into the main memory, and after some time, it retrieves the computed results from its IDB. Note that a similar functioning can be found in common high-performance architectures (e.g., CUDA).

From a technology point of view, the TCL turns hardware tasks into self-contained and closed computing structures, which have access to a limited amount of hardware resources.
(when granted by the HWuK) in an exclusive fashion. Therefore, the hardware tasks can be defined at a high-level of abstraction in both time and area domains as follows.

In the area domain, a task $\theta_i$ is considered to occupy an arbitrarily sized rectangular region on the FPGA, which is defined by its width and height, $h_{x,i}$ and $h_{y,i}$, respectively (see Figure 4.5b). The type and amount of resources used by a task depend on the computation it performs; e.g., a signal processing task will need to use a large amount of DSP48s. The internal architecture of the task, which depends on the location where it was originally synthesised in the FPGA, is described as the succession of the resources it uses column by column, from the leftmost to the rightmost column.

In the time domain, a task $\theta_i$ requires five different phases to complete a computation (see Figure 4.7):

1. During the set-up phase, the task is configured in the FPGA. Previously existing tasks in overlapping positions are deallocated, if any, and a suitable clock signal is routed. In general, the duration of the set-up phase of a task, $t_{A,i}$, is proportional to its size. However, as the amount of time needed to deallocate overlapping tasks cannot be estimated a priori, a worst-case penalty must be added. In any case, we note that the latter time is minimal as a result of using MFWR commands.

2. During the input data delivery phase, the IDB of the task is filled by the HWuK with actual data to be processed. In general, the time required to do so, $t_{D,i}$, is proportional to the amount of data to be loaded. When all input data are copied into the IDB, the HWuK enables the HWS of the task.

3. During the execution phase, the input data is transformed into results by the task’s PEs. The combination of temporarily isolated tasks and hardware-based deterministic computation leads to predictable timing behaviour. Indeed, a task uninterruptedly completes its computation $t_{E,i}$ units of time after it started, regardless of system workload. However, $t_{E,i}$ is not always fixed and known; e.g., iterative calculations with variable number of iterations. In order to deal with these situations the tasks automatically signal their HWS to acknowledge the HWuK when the results are ready in their ODB.
Chapter 4 - Overview of R3TOS and Associated Computing and Dependability Models

4. The **synchronisation phase** refers to the polling process from the HWuK to the tasks’ HWS to detect a computation completion, and spans $t_{S,i}$ units of time. While $t_{S,i}$ is fixed and known for tasks with known $t_{E,i}$; i.e., equal to the time needed to access the HWS once, the synchronisation overhead for tasks with unknown $t_{E,i}$ varies depending on the amount of HWS accesses carried out until the tasks eventually complete their computations.

5. During the **output result retrieval phase**, which spans $t_{R,i}$ units of time, the results computed by the tasks are finally read from the ODB. Note that this phase may be delayed until the results are finally read from the ODB. Note that this phase may be delayed until the results are finally read from the ODB. Note that this phase may be delayed until the results are finally read from the ODB.

![Figure 4.7: Execution phases of a hardware task in R3TOS](image)

When two hardware tasks communicate each other, data must be read from the producer’s ODB prior to being copied to the consumer’s IDB. That is, the **output data retrieval phase** of the producer task $\theta_j$ is immediately followed by the **input data delivery phase** of the consumer task $\theta_i$. Furthermore, the latter two phases are to be preceded by the **set-up phase** of the data consumer task. When merging these three phases with the **synchronisation phase** of the data producer task $\theta_j$, a single ICAP access is formed for each task $\theta_i$ which spans $t_{ICAP,i} = t_{A,i} + t_{S,j} + t_{R,j} + t_{D,i}$ consecutive units of time (see Figure 4.8). During this time, the task is effectively set-up in the FPGA. The fact of grouping together the task phases that need to access the ICAP is beneficial to schedule this limited resource in a more pre-
dictable way, as it is shown in chapter 5. Note that, since \( t_{ICAPI} \) can vary depending on which task precedes \( \theta_i \) in the DAG, it must be dynamically computed with each task release.

As previously introduced in this chapter, R3TOS uses several ways to reduce \( t_{ICAPI} \) towards a higher performance. First, the direct access from a data consumer task to producer task’s ODB results in \( t_{D,i} \) and \( t_{R,j} \) circumvention. Second, the use of DRTs to quickly relocate data between data buffers using DRTs results in reduced \( t_{D,i} \) and \( t_{R,j} \). Finally, the reuse of previously configured tasks results in \( t_{A,i} \) circumvention. In addition, \( t_{E,i} \) can also be reduced by feeding the task with the highest clock rate.

The real-time constraint of R3TOS involves the existence of a relative execution deadline for each task, \( D_i \), which is defined by the application programmer and represents the maximum acceptable delay for that task to finish its execution. The absolute execution deadline for each new task instance, \( d_i \), is computed by adding the task release time, \( r_i \), to its relative execution deadline, \( d_i = D_i + r_i \). Even more important are the relative and absolute set-up deadlines, \( D^*_i \) and \( d^*_i \), which represent the maximum acceptable delay for a task to start the computation in order to meet its deadline. A task is considered to be ready to start computing when it is completely configured in the device and the data to be processed is loaded in its IDB. To achieve the predictability required in real-time, it is always considered the worst-case that any of the aforementioned performance enhancements cannot be exploited.

The set-up deadlines are different for hardware tasks which communicate with other hardware tasks and for those which deliver data to software tasks. This is because the data retrieval phase is included in the model of the data consumer hardware tasks used by HWuK, but it is not in the model of data consumer software tasks used by SWuK. Indeed, the data retrieval operation of a data consumer software task must be invoked from the data producer hardware task itself. Unfortunately, this functioning could interfere with real-time behaviour and thus, a specific model is derived for it. This model is depicted in Figure 4.8. The absolute set-up deadline of a “standard task” which communicates with other hardware tasks, or which receives data from a software task, is equal to \( d^*_i = d_i - t_{E,i} \). This is the case of \( \theta_j \). On the other hand, hardware tasks that deliver data to the main CPU, such as \( \theta_i \), are modelled as two separate “standard tasks” in order to harmonise their management:
An exclusively computing task $\theta_{\text{comp}}$ with $h_{x,\text{comp}} = h_{x,i}$, $h_{y,\text{comp}} = h_{y,i}$, $r_{\text{comp}} = r_i$, $t_{E,\text{comp}} = t_{E,i}$, $t_{\text{ICAP},\text{comp}} = t_{A,i} + t_{R,i} + t_{S,i} + t_{D,i}$, where $\theta_j$ is the data producer of $\theta_i$, $d_{\text{comp}} = d_i - t_{S,i} - 2 \cdot t_{R,i}$ and $d^*_{\text{comp}} = d^*_i - t_{E,i} - t_{S,i} - 2 \cdot t_{R,i}$. Note that the multiplication by two of $t_{R,i}$ accounts for the dual operation of reading data from task’s ODB and copying it to CPU’s IDB.

An exclusively communicating task $\theta_{\text{comm}}$ with $h_{x,\text{comm}} = h_{y,\text{comm}} = 0$, $t_{\text{ICAP},\text{comm}} = t_{S,i} + 2 \cdot t_{R,i}$, $t_{E,\text{comm}} = 0$, $r_{\text{comm}} = t_{SP,\text{comp}} + t_{\text{ICAP},\text{comp}} + t_{E,i}$ and $d_{\text{comm}} = d^*_{\text{comm}} = d_i$.

To achieve the predictability required by real-time behaviour, it is always considered the worst-case execution time $t_{E,i}$ for the tasks. As a result, the HWS must be accessed only once and thus, $t_{S,i} = t_S \forall \theta_i$, where $t_S$ is equal to the time needed to read-back a single frame from the FPGA device. Note that in this situation, HWSs are checked only to confirm the correct ending of tasks’ execution.

Without any loss of generality, it is assumed that there is no a priori knowledge of task release times, which indeed may depend on previous computations; i.e., event-triggered data-dependent computing. Therefore, tasks are considered sporadic and aperiodic.

As shown in Figure 4.9, a hardware task goes through several states during its life cycle. The task is in waiting state until it is triggered. When this occurs, the task switches to ready state, waiting to be granted with ICAP access and assigned a set of FPGA resources.
to start computation. If the task waits for a long time it may miss its deadline, in which case it is discarded, switching back to *waiting state*. When the task is scheduled on time at $t_{SPI} \leq d_i^* - t_{ICAP_i}$, it switches to the *setting-up state*. The task remains in this state while it is configured in the FPGA and also while it is provided with the input set of data to process. While the task performs active computation it is in *executing state*, and it is in *allocated state* when it remains configured in the FPGA after having completed its computation. Note that allocated tasks do not consume dynamic power as they remain in reset state; i.e., their HWS is disabled.

![Figure 4.9: Life cycle of a hardware task in R3TOS](image)

R3TOS keeps track of the state of the tasks at runtime, by grouping them into different task queues: *Ready*, *Executing* and *Allocated*. Note that there is no need for a *Setting-up* queue as only one task can be at this state at any time (i.e., preemption is not allowed).

### 4.2.4 Scheduling and Allocating Hardware Tasks

At runtime, the HWuK needs to decide *when* to schedule and *where* to allocate the tasks buffered in the ready queue (see Figure 4.10). In short, R3TOS selects at every kernel-tick $t_{KT}$ the most suitable ready task to be executed according to both time and area criteria. Note that $t_{KT}$ is configurable by the user depending on the timing-constraints of the application. The fact scheduling and allocation decisions are made online enables data-dependent computation and permits to cope with the unpredictable degradation provoked by spontaneous faults as well as with occasional workload peaks.

As the problem of scheduling and allocating hardware tasks in a 3-D computing space is NP-hard [Steiger et al., 2004], it is not possible to always guarantee time correctness during their execution. Hence, R3TOS targets *soft real-time* performance for hardware tasks. The
HWuK signals the main CPU when any deadline is missed to handle the corresponding timing error based on the premises set by the application programmer, if any. The objective is thus to minimise both the number of missed deadlines and the total lateness in the system.

![Figure 4.10: Scheduling and allocating hardware tasks onto a partially damaged FPGA with R3TOS](image)

The scheduling decisions are vital to meet the real-time execution deadlines and play a key role towards high-performance. Namely, to hide the allocation and data exchange phases of the tasks, to promote multitasking, and to reduce the application execution time. R3TOS uses a non-preemptive EDF porting to schedule access to the ICAP. To compensate for the lack of preemption, which on the other hand is not well suited to be used in RC (see chapter 3), the scheduling algorithm uses a novel way to achieve at the same time a good performance and an adequate real-time behaviour.

R3TOS is one of the first ROS where task allocation acquires a real meaning. In contrast to a slotted reconfigurable system, where the allocation decisions simply consist in deciding in which slot (out of a dozen or less of slots), it is preferable to execute a given task, in R3TOS the tasks can be freely placed within the whole FPGA’s reconfigurable area. This makes the allocation problem more complex as enough adjacent resources must be available in both directions, $x$ and $y$, for allocating a ready task.

As pointed out in chapter 3, the enemy to beat in such a flexible scenario is *external fragmentation*, which is accentuated by the occurring spontaneous faults in the device and, to a lesser extent, by the data traces left by the tasks. R3TOS uses a novel allocation strategy which is aimed at placing the tasks in the locations where they provoke less fragmentation in the device. Note that it is preferable to minimise fragmentation when allocating...
Chapter 4 - Overview of R3TOS and Associated Computing and Dependability Models

the hardware tasks rather than dealing with it once the tasks are running on the device; i.e., defragmentation techniques incur high reconfiguration overhead provoked by extra task relocations.

We make a final remark here on performance. At each scheduling point, \( t_{SP} \), the allocator provides the scheduler with the Maximal Empty Rectangle (MER) to enable it to discard the tasks which cannot be placed in the FPGA early; i.e., \( \forall \theta_i / h_{x,i} \cdot h_{y,i} > MER(t_{SP}) \) are discarded by the scheduler. Note that this saves vital time when making the subsequent allocation decisions.

4.2.5 Executing Hardware Tasks

So far we have seen that HBC tasks are connected together to speed-up the transmission of intermediate partial results among them, i.e., if possible the consumer task accesses its input data directly from the ODB of the producer task. While this scheme improves the performance by minimising ICAP time for carrying out inter-task communications, it also constrains multitasking capabilities. As shown in Figure 4.11, only half of the HBC tasks in a computation chain can perform active computation simultaneously (stripped in this figure) as only one task can access the shared data buffers at each time. This restriction does not apply in the case of LBC tasks, for which multitasking capabilities remain untouched as data buffers are not shared. However, as there is only one ICAP per FPGA, both LBC and HBC tasks must be sequentially enabled to start computing (i.e., access to HWS is sequential), and thus multitasking is limited by the number of tasks that remain still executing when subsequent tasks are enabled.

4.2.6 Input / Output

Besides interacting with memory-mapped peripherals by means of its main CPU, so far we have seen that R3TOS can also deal with streaming data I/O. To process high data-rates (i.e., hundreds of Mbps), static logic must be used. However, dynamic hardware tasks can be used to improve efficiency in low data-rate applications. As shown in Figure 4.12, R3TOS buffers together the data samples received through each data channel to form large data-streams. Note this can be done by the main CPU itself when the data-rate is very low or using dedicated logic when the data-rate is moderate. In the former case data is stored in the main
Figure 4.11: Limited multitasking in a chain of HBC tasks. D refers to data to be processed, P to intermediate partial results computed by the tasks and R to the final results computed by the chain.
memory while in the latter case data is stored in specific interface data memories which can be directly accessed by the HWuK. This frees up the CPU from having to handle and/or process the received data. Unfortunately, the interface logic is application-dependent and hence, it is not amenable to systematic generalisation.

The hardware tasks in charge of processing data are triggered only when the size of the streams is sufficiently large; e.g., when the data buffers are full. These tasks exploit data-parallelism and use custom deep pipelines to process the streams at hardware speed. Indeed, it is mandatory that data processing speed is significantly greater than data input rate; e.g., a 16 KHz voice signal processed by a hardware filter running at 100 MHz. At the output, R3TOS releases the computed results sample by sample at the rate required. Note that to hold a continuous flow of data, real-time behaviour is necessary.

### 4.3 The R3TOS Dependability Model

R3TOS brings closer to modern FPGA-based systems some of the well accepted concepts in dependability theory (e.g., [Avizienis et al., 2004]).

#### 4.3.1 Dependable Theory

As shown in Figure 4.13, a *fault* (e.g., SEU, EM, HCI, TDDB) produces an incorrect state in a component (e.g., a wrong data value in a memory or register). The incorrect state of the component is called *error* and, may provoke a deviation in the behaviour of a system when it is propagated to the system output. This deviation, named *failure*, can occur either in the value domain, e.g., an incorrect value is presented at the component interface, or in the time domain, e.g., the value is presented outside the intended interval of (real-) time. The *Mean Time To Manifest (MTTM)* is defined as the elapsed time interval between the occurrence of the error and the instant when it is propagated to the output of the system, and varies depending on the functionality assigned to the erroneous data/resource. An error can also be *wiped-out* if it is overwritten with a new value before causing a failure. The new value can be automatically written by the computation (e.g., in a pipeline processing stage), or intentionally by some error-correction mechanism (e.g., scrubbing). Indeed, the tests carried out by Xilinx Inc. have shown that as little as 2% to 10% of the total soft-errors occurring in their FPGAs lead to a system failure in a typical application [Schumacher, 2012].
Figure 4.12: Dealing with moderate data-rate applications using R3TOS
**Chapter 4- Overview of R3TOS and Associated Computing and Dependability Models**

*Time To Detect (MTTD)* is defined as the elapsed time interval between the occurrence of the error and the instant when it is found out that the system functioning deviates from the expectation. The *error-detection coverage* refers to the probability that an error is detected and depends on the ability of the diagnostic subsystem.

![Diagram of dependability threats](image)

**Figure 4.13:** The fundamental chain of dependability threats

A fault-tolerant system is able to detect, isolate and recover from spontaneously occurring faults before they lead to failure. To develop these capabilities, it is very useful to build fault-tolerant systems out of components that are *Fault Containment Units (FCUs)*. A component is an FCU if the direct effect of a single fault affects only the operation of that component [Lala and Harper, 1994, Kopetz, 2011]. To ensure this, both physical and logical isolation are mandatory. The physical isolation requires that no common resources are shared among the components and the logical isolation requires that no data produced by a fault-affected component is propagated to other components in the system; e.g., by means of an erroneous message. To achieve this, error propagation boundaries must be included around each FCU.

In order to tolerate the failure of an FCU, replicated FCUs are grouped into *Fault Tolerant Units (FTUs)*. Despite the fact that a single failure of an FCU can be masked by the rest of redundant components within the FTU, permanent failures of the FCUs reduce or eliminate any further fault-masking capability. These failures must be detected and reported to force the reparation or replacement of the faulty components in the next scheduled maintenance service. Finally, *fail-safe shutdown* refers to the process of bringing the system to a safe state when it has failed (e.g., in the case of a railway signaling system, the safe state could be to set all signals to red and thus stop all the trains [Kopetz, 2011]).

### 4.3.2 R3TOS Contributions to Dependability

This section covers the main contributions to dependability made by R3TOS as a result of its flexible low-level management of FPGAs resources. It also presents the fault hypotheses;
i.e., assumptions that are made about the types and number of faults that must be tolerated by R3TOS. This section, however, does not get into the details of dependability issues related to the main CPU.

**Physical Isolation**

R3TOS enhances the physical isolation among the hardware tasks. Namely the tasks do not share neither logic nor routing resources, and they share limited clocking resources. Indeed, the implementation flow used in R3TOS leads to a true physical separation for the tasks, which are self-contained into mutually exclusive FPGA regions that are dynamically assigned by the HWuK. In addition, R3TOS exploits the branched clock-tree available in modern Xilinx devices to feed each task with a separate clock signal and even to recover from failed clock sources. As a result, the hardware tasks can be considered quasi-FCUs which only share the power supply of the chip and the global clock buffer.

This is an improvement with regard to other approaches, where despite the fact that system components are placed in separate regions (see Figure 4.14a), some of them share the same routing resources (see Figure 4.14b). In this scenario, a single fault affecting one of the shared routing resources could provoke a shortcut between two or more routes belonging to two or more different components, resulting in collective failure [Bellato et al., 2004]. Indeed, in order to deal with this problem, Xilinx Inc. has recently launched the Isolation Design Flow (IDF), which allows for physical separation of different components on the chip [Corbett, 2012, Gantel et al., 2012]. In effect, IDF defines a boundary, of at least one CLB-wide horizontally and vertically, where the use of both logic and routing resources is prohibited. Notably R3TOS provides this support at runtime, as the tasks are executed on the FPGA. Besides, in IDF, connections between isolated regions pass through direct trusted routes. In R3TOS this problem is solved by using either “not physical” ICAP-based communication channels, or DRTs which are treated as standard tasks guaranteeing the physical isolation (see chapter 6).

In connection with IDF, Xilinx Inc. has released the Isolation Verification Tool (IVT), which is an independently implemented design rule checker that marks with an 'X' the FPGA resources where isolation violations are detected (see Figure 4.15).
Figure 4.14: Physical isolation violations in a NoC implementation with 3 components and 4 routers [Azkarate, 2012]

Figure 4.15: IVT Floorplan detail showing isolation violations among two components, Counter1 and Counter2 [Corbett, 2012]
Logical Isolation

The HWuK is an effective error propagation boundary as all inter-task communications pass through it. In effect, there are no other channels through which hardware tasks can communicate with each other (e.g., shared memory). Even when an ODB is used as IDB of another task, or when data is relocated using DRTs, R3TOS can check the correctness of the data temporarily stored in the buffer prior to proceeding to configure the data consumer task. R3TOS is thus able to detect and discard erroneous sets of data, either in the value domain (e.g., redundant instances of the same task do not produce the same results) or in the time domain (e.g., a HWS is not activated when expected).

Special cares must be taken to ensure that the HWuK does not corrupt hardware tasks while carrying out inter-task communications or when configuring new tasks on the device. In fact, any resource in the FPGA could be unintentionally corrupted by the HWuK itself when accessing an erroneous position in the configuration memory. To protect against these errors, R3TOS includes a Configuration Guardian (CG) which handles accesses to the FAR register in the configuration interface. The latter exclusively allows access to the resources assigned to a given task to configure it or to communicate with it. Besides, the CG prohibits access to the resources assigned to HWuK except when the privileged mode is enabled (e.g., to scrub a dormant soft-error affecting the HWuK). The HWuK contains thus all the functionality that must be privileged to access the FPGA configuration port (e.g., task control and inter-task communications and synchronisation), while the rest of R3TOS functionality is implemented by the main CPU without direct access to the FPGA configuration port. Therefore, the CG conceptually acts as the Memory Management Unit (MMU) in a conventional processor, playing a vital role to isolate the hardware tasks in the configuration domain.

Note that this centralised functioning improves the composability and eases the certification of R3TOS-based systems sustained in a simple, reusable and trusted (i.e., free of design faults) implementation of the HWuK [Heiser et al., 2007]. Indeed, a certified implementation of the HWuK should be saved as a hard-macro to be instantiated in every R3TOS-based design, where the hardware tasks that implement the application-dependent functionality could be individually verified based on the fact that their interactions are im-
plicated by the HWuK. That said, the HWuK constitutes a single source of failure, which must be carefully protected against faults through special hardening (e.g., TMR).

**Redundancy**

R3TOS is capable of executing redundant instances of the same (critical) hardware task in parallel at distinct positions within the FPGA (spatial redundancy) or at different times (temporal redundancy) to build quasi-FTUs. In effect, as long as meeting the real-time constraints, different instances of the same task can be executed until achieving the desired reliability level in the result (i.e., repetitiveness). Based on the fact that the execution time of the tasks is likely to be substantially shorter than fault rates, it is assumed that only one of the redundant instances of a task may be affected by a fault ($t_{E,t} + t_{ICAP,t} < \text{fault_rate}$).

As shown in Figure 4.16, the FPGA die is divided into three Fault Containment Computation Regions (FCCRs), where each of them is to host one instance of the same task. However, note that to build FTUs in the stricter sense, the FCCRs must be distributed across different FPGA chips.

The FCCRs are separately managed by the allocator, i.e., each task instance is individually allocated within each FCCR in order to circumvent the specific damaged resources in that FCCR. In this context, the scheduler is provided with three MERs, each describing the state of one FCCR. Based on the criticality of the tasks and the MERs, the scheduler can
choose between: (1) execute a single instance of a given (non-critical) task when the task fits only in one FCCR, (2) execute two instances of a (critical) task when it fits in two FCCRs (i.e., lockstep), or (3) execute three instances of the (critical) task when it fits in all the FCCRs (i.e., TMR). Lockstep permits to detect computation errors when both instances come up with different results, but the task must be executed another time to determine the correct result. Therefore, this method is not amenable to be used in real-time R3TOS. On the other hand, TMR does not only allow to detect computation errors, but also to recover from single errors.

To simplify and accelerate the result voting process, R3TOS is equipped with a CRC32 module to compute the 32-bit CRC over each set of results read through the ICAP at the data rate; i.e., when accessing the ODBs of data producer tasks. The CRC values associated with the data sets produced by redundant instances of the same task are compared to detect computation errors and, if any, to determine which instance is giving the erroneous result.

We acknowledge that erroneous results could still produce correct CRCs. However, this is a good way to reduce the complexity of dealing with large amounts of data (e.g., the 18 Kb stored in a BRAM) which are represented by only one 32-bit value. Otherwise it is necessary to temporarily buffer the three sets of data and use a more complex voter which could itself introduce error.

Figure 4.17a shows the time diagram of the execution of the three redundant instances of a critical hardware task $\theta_i$ in R3TOS. First, the HWS of the three instances of the data producer task $\theta_j$ are checked to make sure all of them have finished correctly their computation. If so, the results computed by $\theta_j$ are accessed through the ICAP and the resulting CRC values are voted. Note that the third data access can be circumvented when the first two CRC values match. Also note that there is no need to vote again if another task needs to use the results computed by $\theta_j$ in the future; i.e., the three redundant instances of the same task lead to a unique data trace after having performed the voting process.

In order to speed-up the configuration of the three redundant instances of critical tasks, MFWR configuration commands are used. This is represented with concurrent configuration phases in Figure 4.17a. Two remarks are in order here. First, the partial bitstreams generated using Xilinx tools must be decomposed into individual frames to enable the use of MFWR commands (see Figure 4.17b). Note that the data of each configuration frame can
be easily known by enabling the bitstream debugging option in the Xilinx Bitgen tool. Second, the three redundant instances of the tasks must be allocated on the same half of the FPGA to fully exploit MFWR-based acceleration, otherwise the configuration data stored in the MFWR register must be reversed incurring time overheads. Note that the partition of the FPGA into FCCRs depends on the amount and type of critical tasks in the application and on the layout of the used FPGA device. Moreover, all of the task instances must be fed with the same regional clock lines, otherwise the clocking configuration must be individually adjusted for each task instance during the set-up phase.

Furthermore, MFWR commands are also used to speed-up the broadcast of the (same) data to the redundant instances of a critical task. However, the total number of read-back
frames from the FPGA is double because MFWR commands operate on single frames and, each frame read-back operation includes an extra pad frame. Therefore, in total 4 or 5 accesses to \( \theta_j \)’s ODB are necessary: the first 2 or 3 to vote the correct results and another one, which is indeed double due to the pad frame, to access the correct data.

Redundancy is handled in a layered manner. The configuration manager adjusts the FAR address of the MFWR commands in the bitstream headers to allocate the redundant task instances on the desired FPGA locations. Besides, it votes the results computed by the redundant task instances and presents the computation layer with the correct ones. At a higher abstraction layer, the allocator hides the details of the FCCRs to the scheduler, which indeed can manage the quasi-FTUs as standard tasks with some modified parameters.

As shown in Figure 4.17a: 

\[ t_{FTU, ICAP} \approx t_{A,i} + (2 \text{ or } 3) \cdot t_S + (4 \text{ or } 5) \cdot t_{R,j} + t_{D,i} \text{ and } t_{FTU, E} = t_{E,i}; \]

and for an exclusively communicating task 

\[ t_{FTU, ICAP, comm} \approx (4 \text{ or } 5) \cdot t_{R,i} + (2 \text{ or } 3) \cdot t_S. \]

**Error-Detection Coverage**

The error-detection coverage of R3TOS is based on three methods. First, redundancy to detect and mask computation errors. Second, read-back and scrubbing to detect and correct configuration errors. Third checking of the HWSs to detect temporal errors. The fault-handling strategy used by R3TOS is depicted in Figure 4.18.

The resources assigned to a task that has computed an erroneous result are kept in quarantine while an exhaustive diagnostic test is carried out on them. The objective of this test is to detect and prevent the future use of damaged resources, if any.

While the aforementioned diagnostic test is performed only when a computation is erroneous, due to the criticality of the HWuK, its configuration state is periodically checked, using Frame_ECCs, to correct any errors before they lead to system failure. If the configuration error is the result of a damaged resource, the system initiates a fail-safe shutdown. In addition, all the HWuK memories are to be protected by means of ECCs to automatically correct any upset affecting them. However, it must be noted that in the current R3TOS implementation this has not been done yet.

To detect functioning failures, the HWuK is equipped with some monitors to supervise the interactions between its internal components (e.g., scheduler, allocator and configuration manager), and with the aforementioned CG to trap undesired access to the FPGA con-
Chapter 4 - Overview of R3TOS and Associated Computing and Dependability Models

![Diagram of fault-handling strategy used in R3TOS](image)

**Figure 4.18:** Simplified fault-handling strategy used in R3TOS

The key benefit of R3TOS is its ability to exploit the underlying reconfigurability of the FPGA chip. Specifically, continuous task reconfiguration prevents the accumulation of soft-errors and the permanent damage in the chip is conveniently circumvented when allocating tasks as long as it does not affect the HWuK itself.

When it is detected a malfunctioning of the HWuK, a full FPGA configuration is forced by toggling the PROG_B pin\(^4\) after having disabled the ICAP interface. Note that this results in the loss of all computations currently executing, but on the other hand, it is a highly reliable mechanism which exclusively uses FPGA's built-in configuration circuitry. Moreover,

\(^3\)In one-hot codes there is a single high ('1') bit and all the others low ('0').

\(^4\)PROG_B pin is an active-low asynchronous full-chip reset.

\(^5\)A similar error recovery mechanism is available in other Xilinx FPGA families (e.g., Spartan-3/6).
the configuration information is accessed directly from a non-volatile memory, which is assumed to be non-corruptible. As soon as R3TOS is recovered, the configuration state of the HWuK is checked to determine whether the error was provoked by a damaged resource. If this is the case, the system initiates a fail-safe shutdown.

4.3.3 Fault Hypotheses in R3TOS

The following presents the fault hypotheses of R3TOS:

- The hardware tasks form quasi-FCUs as they are physically separated into disjoint positions in such a way that they only share the power supply and limited clocking resources.

- A configuration guardian isolates hardware tasks in the configuration domain.

- Redundant instances of the same hardware task are executed in different FCCRs. It is assumed that only one of the task instances may be affected by a fault due to the dynamic nature of the tasks ($t_{E,i} + t_{ICAPI} \ll fault\_rate$).

- Configuration errors are corrected in the configuration memory.

- A diagnostic test localises damaged resources within the FCCRs and the damaged resources are never used again.

- In response to any permanent damage affecting the HWuK the system executes a fail-safe shutdown. Note that the probability that a resource used by the HWuK gets damaged is reduced as it uses a small portion of the FPGA resources.

4.4 R3TOS General Architecture

Figure 4.19 shows the general block diagram of R3TOS, which basically comprises three main parts: HWuK, the main CPU and memory. The interface of the HWuK is standard enough to ensure compatibility with the most common on-chip / off-chip processors and memories. External memory is necessary when developing large reconfigurable applications because of the limited availability of on-chip storage resources in current FPGAs, but it is expected that R3TOS can be implemented on a single FPGA chip in the future.
Figure 4.19: R3TOS block diagram (clock signals are not shown)
**R3TOS Hardware MicroKernel (HWuK)**

The HWuK includes the configuration manager and two servers which run upon it: scheduler and allocator. Each component is separately implemented to enable parallelism in the execution of the HWuK processes; i.e., while the scheduler manages the state task queues, the allocator can compute the MER, the configuration manager can decode a Frame_ECC syndrome, and a task can be configured on the FPGA. The parallel cooperation of simple components does not only result in low runtime overhead but also in acceptable area overhead; i.e., the main core of all HWuK components is a tiny Xilinx PicoBlaze, which requires only 96 slices in a Virtex-4 FPGA. Note that this architecture promotes the core spirit of a microkernel: **upgradability**, e.g., the allocator and scheduler servers can be updated to run more efficient algorithms which might be designed in the future without having to modify the rest of components in the HWuK, and **scalability**, e.g., multiple instances of the allocator can be used to speed-up the allocation process in very large FPGAs.

The cooperation among the HWuK components is mastered by the scheduler, with the allocator and the configuration manager acting as slaves. The communication between the components follows a very strict set of rules which are supervised by two monitors. Each monitor is thus capable of detecting any malfunctioning in each pair of communicating components; i.e., scheduler-allocator and scheduler-configuration manager.

The internal architecture of the HWuK components is structured around the PicoBlaze core. The PicoBlaze executes an optimised assembly program which is based on interruptions to reduce the response time, relying on an interrupt controller to handle the interruptions. Furthermore, each PicoBlaze uses a dedicated data BRAM to store the information associated with the corresponding HWuK process(es) it executes. Hence, the scheduler manages the task queues in the task BRAM, the allocator keeps track of the available resources on an FPGA BRAM (state BRAM), and the configuration manager executes pre-defined sequences of configuration commands from a bitstream BRAM. The fact that these memories are dual-ported is conveniently exploited. For instance, the configuration manager can mark any detected damaged resource as non-usable directly in the FPGA state BRAM, and the main CPU can set in ready state any triggered task directly in the task BRAM.
Specific for the scheduler is a timer to generate the kernel ticks. Additionally, the kernel timer supervises the correct functioning of the scheduler acting as a watch-dog timer. The scheduler's PicoBlaze must generate at least one alive pulse within a maximum number of kernel ticks. Indeed, it is crucial for the reliability of the system to monitor the state of the scheduler as it is the master in the HWuK.

Specific for the allocator are three coprocessors: (1) an Architecture Checker (AC) to speed-up the search of feasible allocations where the FPGA layout is compatible with the internal architecture of the tasks, (2) an Empty Area Descriptor Updater (EADU) to accelerate the intermediate computations required by the allocation algorithm, and (3) an Allocation Quality Evaluator (AQE) to accelerate the making of the allocation decisions.

The configuration manager interacts with the configuration-related built-in logic included in the FPGA. Notably, it is equipped with an FSM to drive the ICAP at the highest allowed clock frequency, and with a CRC32 module to compute the 32-bit CRC over a set of data read from the ICAP. Besides, the configuration manager interacts with the Frame_ECC logic to detect and localise upsets in the configuration frames. Finally, the configuration manager has access to the STARTUP primitive with the objective of initialising the flip-flops of the hardware tasks with a predefined value (i.e., INIT values). Since this is a very critical primitive which allows accessing internal signals to the configuration logic of the FPGA, i.e., Global Set/Reset (GSR), specific support logic is added to guarantee its safe functioning. The STARTUP primitive has been used due to experienced problems when attempting to use configurations commands to initialise the flip-flops. On the other hand, no problems have been encountered when using configuration commands for capturing the state of the flip-flops (i.e., GCAPTURE operation).

As can be seen in Figure 4.19, the HWuK also includes some extra functionality distributed along the device. This includes TCLs, attached to the hardware tasks, and the circuitry to manage and diagnose the clocking resources, which is implemented next to the rightmost and leftmost IOB/BUFR columns.

**R3TOS Main CPU**

In the current R3TOS implementation a Xilinx on-chip processor, namely a 32-bit MicroBlaze soft-core, is used as the main CPU. The MicroBlaze is coupled with a set of peripherals
which provide additional functionality (e.g., timer or interrupt controller), connection to the external world (e.g., RS232 serial line or Ethernet), or which increase the performance (e.g., DMA). The peripherals are interconnected by means of an On-chip Peripheral Bus (OPB). The interface with the HWuK is based on interruptions and shared memory (i.e., task BRAM and IDB/ODB).

The program executed by the main CPU is held in a directly accessible program memory. In the specific case of the MicroBlaze this memory is implemented using dual-ported BRAMs and thus, it must be disabled and its clock must be stopped by the HWuK prior to accessing the content of any BRAM within the FPGA. Otherwise, the program code executed by the MicroBlaze could get corrupted. Despite this displeasing effect does not appear when using other processors which do not use dual-ported BRAMs to store their program, the clock stopping capability is still required when using the STARTUP primitive. Indeed, while the GSR signal is active, the BRAMs cannot be correctly accessed and thus, any processor relying on BRAMs to store its program should be stopped to prevent executing undesired instructions.

FIFO-based high speed communications between the IDB, ODB and MicroBlaze are achieved by connecting them using Fast Serial Links (FSLs). In addition, the IDB and ODB are also accessible through the aforementioned OPB bus to allow individual access to specific positions.

Memory

In order to reduce the amount of components in the R3TOS system, both the bitstream memory and the main memory are implemented using a single external memory chip, despite the fact they are conceptually independent components. More specifically, the external memory chip is used to store: (1) the data and code segments of the software tasks, (2) the data segments and bitstreams of the hardware tasks and, and (3) the bitstreams of the DRTs. Moreover, there is a Pointer Table (PT) located in the lowest part of the memory, which is used by the HWuK to know the exact location of each task bitstream. Namely, the bitstream of a task, or DRT, with identifier TaskID starts in the position specified by the pointer located at position TaskID, and it ends in the position specified by the pointer located at position TaskID+1; i.e., \(PT[\text{TaskID}] \rightarrow PT[\text{TaskID}+1]-1\). Aiming at achieving
the highest performance when accessing the external memory, a custom memory controller with capability to deal with the pointers stored in the pointer table is included in the HWuK.

In Figure 4.19 note that the white parts of the memory are accessed exclusively by the HWuK while the black parts are accessed only by the main CPU. The most typical case of a single port memory is assumed in the current R3TOS implementation and hence, HWuK includes an arbiter to coordinate access from the HWuK itself and from the main CPU.

4.5 Chapter Conclusion

This chapter introduced the bases of R3TOS, a novel reliable reconfigurable real-time operating system aimed at putting the versatile resources embedded in modern FPGAs at the service of computation in a reliable way, promoting direct translation of electronic advances into system performance improvement. By sharing FPGA resources in time, R3TOS is able to execute very large applications in small devices, harnessing the maximum performance of the available silicon and reducing the dynamic power consumption.

While one of the key features of R3TOS is its ability to manage FPGAs resources at low-level, it presents the user with a set of high-level OS-like mechanisms and services (e.g., task relocation and data exchanges among tasks), which abstract low-level related issues and ease the development of reconfigurable applications. In effect, R3TOS provides support to seamlessly deal with both software and hardware tasks when programming reconfigurable hardware.

Central to R3TOS is real-time performance and fault-tolerance. While both software and hardware tasks are scheduled based on real-time constraints, software tasks are executed in a main CPU and hardware tasks are dynamically mapped to non-damaged FPGA resources and individually provided with a reliable clock source of the highest usable frequency by each task. Damaged resources are located by means of a self-diagnosis mechanism which normally does not interfere with the normal functioning of the system. Besides, the continuous reconfiguration of hardware tasks limits the accumulation of soft-errors in the configuration memory of the FPGA, and the mutual-exclusive mapping of tasks combined with redundancy ensures the highest dependability. As a result, device lifetime is to be prolonged and the impact of aging and other reliability threats on correctness and performance is minimised.
As introduced in chapter 4, the R3TOS HWuK includes a scheduler and an allocator to manage the workload and system resources. Indeed, this is the traditional way to decompose the 3-D computing space defined by partially reconfigurable FPGAs into two simpler and more manageable domains: the 1-D *Time domain*, to be handled by the scheduler, and the 2-D *Area domain*, to be handled by the allocator.

However, keeping in mind the original 3-D problem, it seems reasonable either to consider some area notions when making the scheduling decisions or to include some time aspects when making the allocation decisions. This chapter explores these two alternatives to come up with a set of novel scheduling and allocation algorithms. The proposed algorithms are compared against each other and with the state of the art by means of experimental simulations. Based on the obtained results, the most effective and efficient solution to be finally implemented in the R3TOS HWuK can be decided upon. Unlike most related work in the field, which usually does not go beyond simulations, R3TOS HWuK implements the algorithms on real FPGA hardware. This is possible because our algorithms consider the limitations imposed by currently available FPGAs (see chapter 2), e.g., ICAP sequentialness and lateness, as well as most of the particularities of RC (see chapter 3), e.g., inter-task dependencies.

---

Most of this chapter has been published in [Iturbe et al., 2013a].
5.1 Real-Time Scheduling Algorithms

Scheduling real-time hardware tasks in partially reconfigurable FPGAs is a relatively new problem. The work done up to date inherits the mono-core processor state of the art, where EDF dominates the rest of scheduling algorithms [Danne and Platzner, 2005, Dittmann and Frank, 2007]. Indeed, there is a similarity between CPU access in mono-core processors and ICAP access in FPGAs: the access is exclusive and sequential. Based on this fact, Dittmann and Frank proposed to apply EDF to schedule the hardware tasks based on their absolute set-up deadlines [Dittmann and Frank, 2007]. They reported better results when using the set-up deadlines \((d_i^s)\) to dispatch the tasks than when using the computation deadlines \((d_i)\), especially when the amount of time the tasks need to access the ICAP \((t_{ICAP,i})\) is in the range of their execution time \((t_{E,i})\). This is particularly interesting in the scope of R3TOS, where the ICAP is used not only to configure the tasks but also to carry out inter-task communications and synchronisation. Moreover, the low computational burden of EDF is also very attractive, i.e., scheduling decisions are nearly immediate provided the tasks are sorted in the ready queue by increasing set-up deadlines.

A major difference between the RC scenario targeted by Dittmann and Frank and R3TOS is that, while the former considered a regular slot-based 1-D system with a fixed amount of equally sized and shaped tasks running on it, the latter considers a flexible 2-D area model with a runtime variable amount of tasks, each with its own size and shape (see Figure 5.1). As a result of the different size of the tasks, R3TOS must not only deal with different allocation times for them, but also with external fragmentation in the device. This contrasts with Dittmann and Frank’s RC scenario, where every scheduled task can be allocated within the same amount of time, and exactly to the same resources that were previously assigned to another task. Moreover, Frank and Dittmann targeted hard real-time performance, while R3TOS targets soft real-time; i.e., keep the performance as good as possible while missing the least amount of deadlines. This is why R3TOS does not execute any acceptance test when a new task is released, it will simply serve as many tasks as possible.

Preemption deserves special attention in RC due to its inner computing particularities and the fact there are still no efficient techniques and methods available to do it. Preempting a task during its execution phase is discarded due to the excessive reconfiguration overhead.
it provokes. On the other hand, preempting a task which has not started computing yet is admissible (i.e., no context saving is needed), but this comes at a price. In a general sense, a task allocation can be preempted only at pre-defined points (see chapter 2), namely when all the configuration words specified in the packet headers of task’s partial bitstream are configured. This results in unpredictable delays which could conflict with real-time performance. Even worse, in the specific case of bitstreams which do not use compression, it is likely that there is a single packet that includes all the configuration information, making preemption unfeasible.

Moreover, in contrast to what occurs in a mono-core processor, ICAP-time is not synonymous with computing time. A task starts computing only when it is completely set-up in the FPGA and the data to be processed is loaded in its Input Data Buffer (IDB). Therefore, preemption does not make EDF optimal as it does in mono-core processors. Indeed, it may lead to deadline miss when the FPGA resources are occupied for a long time without performing active computation. This is shown in Figure 5.2: $\theta_3$ misses its deadline as it is delayed too long due to the preemption of $\theta_1$ and the subsequent contention in the access of the FPGA resources this provokes.

Arguing that non-preemptive scheduling algorithms are easier to implement than preemptive ones, and based on the foundations presented in chapter 4, which pursue the most efficient use of FPGA resources, R3TOS does not use preemption. The objective is thus to execute the tasks as fast as possible to provoke the least contention when accessing the FPGA resources.
Chapter 5- R3TOS Algorithms to Manage Workload and FPGA Resources

(a) Preemptive EDF: $\theta_3$ misses its deadline

(b) Non-preemptive EDF: Feasible scheduling

Figure 5.2: Preemptive vs. non-preemptive EDF scheduling
A preliminary porting of Dittmann and Frank’s non-preemptive EDF algorithm to the reconfigurable scenario targeted by R3TOS is shown in Algorithm 5.1. This porting includes the capability to discard the tasks with greater area than the area available in the device early (see line 3), which is an easy but effective way to account for external fragmentation in the scheduler. Hence, the Allocate() function is only invoked when the area of the task to allocate is smaller than the Maximal Empty Rectangle (MER). This is the basis for the scheduling algorithm presented in this thesis, the Finishing-Aware EDF (FAEDF), which includes the capability to “look ahead” to find future releases of adjacent pieces of area when executing tasks finish. This capability is designed to replace task preemption in the cases when preemption would be beneficial.

\begin{algorithm}
\begin{algorithmic}
\State \textbf{input} : (1) List of R ready tasks, sorted by increasing $d^*_i - t_{ICAP,i}$
\State \hspace{1cm} (2) MER, given by the allocator
\State \textbf{output}: Scheduled Task
\State $i \leftarrow$ First Task in Ready Queue sorted based on Time;
\While{$i \leq R$}
\State \textbf{if} $MER \geq h_x \cdot h_y, i$ \textbf{then}
\State \hspace{1cm} \textbf{if} Allocate($i$) $\neq \emptyset$ \textbf{then}
\State \hspace{2cm} \textbf{return} $i$;
\State $i \leftarrow$ Next Task in Ready Queue;
\State \textbf{return} $\emptyset$;
\EndWhile
\end{algorithmic}
\caption{Porting of Dittmann and Frank’s EDF algorithm to R3TOS: Schedule_EDF()}
\end{algorithm}

5.1.1 Finishing Aware Earliest Deadline First (FAEDF)

When using FAEDF, a ready task $\theta_i$ which cannot be allocated on the FPGA at a given scheduling point $t_{SP}$ is only discarded if there are not any sufficiently large tasks finishing before its deadline expires. If any, the allocation of $\theta_i$ is delayed until the executing task(s) finish and there are enough free resources in the device. The time left until then is used to schedule other ready tasks which can be completely set-up before the deadline of $\theta_i$ expires. Note that despite these tasks are also selected based on EDF, the scheduling policy is altered as tasks with farther deadlines can be scheduled before others with closer deadlines, but which do not meet the aforementioned set-up time requirement. In case there are no ready tasks which meet the set-up time requirement, FAEDF does not schedule any task (i.e., it produces a “blank schedule”), assuming the expected finish of the executing tasks is close enough.
During the scheduled blank times other R3TOS services which need to access the ICAP could be executed; e.g., the configuration state of the HWuK could be checked in the configuration memory. Finally, not occupying the ICAP when the real-time constraints are loose allows to rapidly allocate any incoming task with very tight deadline.

The fact of using ICAP-time for performing other operations rather than task configuration may seem contradictory. However, blank times are only scheduled when the real-time constraints of the ready tasks are loose, i.e., when $\sum_{\forall k \in \text{Ready Queue}} \frac{t_{ICAP,k}}{d^*_k - t}$ is low, with $t$ referring to the actual time in the system. Otherwise, it is assumed that the fact of altering the EDF policy to give a “second chance” to a specific ready task to meet its deadline may lead to miss more deadlines and thus, it is discarded. That is, FAEDF proceeds as standard EDF when the real-time constraints of the ready tasks are tight, i.e., when $\sum_{\forall k \in \text{Ready Queue}} \frac{t_{ICAP,k}}{d^*_k - t}$ is high.

The task-set shown in Table 5.1 is used to illustrate the improvement brought about by FAEDF’s “look ahead” capability with regard to non-preemptive EDF. As shown in Figure 5.3a, scheduling $\theta_3$ at $t_{SP,B}$ delays the subsequent allocation of $\theta_2$ too long, and as a result, the latter misses its deadline. Note that $\theta_2$ cannot be allocated at $t_{SP,B}$ because there is no enough adjacent area in the FPGA. On the other hand, as shown in Figure 5.3b, at $t_{SP,B}$ FAEDF finds out that $\theta_2$ can be allocated later using the resources that will be released by $\theta_1$. The time until then is used to allocate $\theta_4$. After having allocated $\theta_4$, a blank schedule is produced because $\theta_3$ cannot be completely set-up in the remaining time until $d^*_2 - t_{ICAP2}$. Note that if $d^*_3$, which is met just in time in Figure 5.3b, had been tighter, FAEDF would have functioned as standard EDF. Namely, it would have scheduled $\theta_3$ at $t_{SP,B}$, assuming that it is impossible to meet at the same time both $d^*_2$ and $d^*_3$.

<table>
<thead>
<tr>
<th>$h_{x,i}$</th>
<th>$h_{y,i}$</th>
<th>$t_{ICAP,i}$</th>
<th>$t_{E,i}$</th>
<th>$D^*_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta_1$</td>
<td>4</td>
<td>4</td>
<td>6 $\cdot$ $t_{KT}$</td>
<td>3 $\cdot$ $t_{KT}$</td>
</tr>
<tr>
<td>$\theta_2$</td>
<td>4</td>
<td>2</td>
<td>4 $\cdot$ $t_{KT}$</td>
<td>9 $\cdot$ $t_{KT}$</td>
</tr>
<tr>
<td>$\theta_3$</td>
<td>2</td>
<td>3</td>
<td>4 $\cdot$ $t_{KT}$</td>
<td>8 $\cdot$ $t_{KT}$</td>
</tr>
<tr>
<td>$\theta_4$</td>
<td>2</td>
<td>2</td>
<td>2 $\cdot$ $t_{KT}$</td>
<td>9 $\cdot$ $t_{KT}$</td>
</tr>
</tbody>
</table>

Table 5.1: Task-set used in Figure 5.3 (Note: $t_{KT}$ is a kernel tick)

Algorithm 5.2 shows FAEDF’s pseudocode. Lines 2 to 6 are exactly the same EDF algorithm presented in Algorithm 5.1, and lines 7 to 23 are the “look ahead” extension to it. This extension is enabled only when the real-time constraints are slacker than a pre-defined
Chapter 5 - R3TOS Algorithms to Manage Workload and FPGA Resources

(a) Non-preemptive EDF: $\theta_2$ misses its deadline

(b) FAEDF: Feasible scheduling

Figure 5.3: EDF and FAEDF scheduling
Chapter 5 - R3TOS Algorithms to Manage Workload and FPGA Resources

Threshold, which typically ranges from 0.6 to 1 (see line 7). The executing queue is searched at lines 10 to 13 to find an executing task $\theta_j$ which would release a sufficiently large amount of resources when it finishes to allocate the ready task $\theta_i$ ($h_{x,j} \geq h_{x,i}$ and $h_{y,j} \geq h_{y,i}$). If any, the ready queue is then scanned at lines 16 to 22 to find a ready task $\theta_m$ which can be completely allocated before $d_i^*$ expires ($t + t_{ICAP,m} \leq d_i^* - t_{ICAP,i}$). The scheduler produces a blank scheduling when it is possible to allocate $\theta_i$ using the resources to be released by an executing task $\theta_j$, but there are no $\theta_m$ ready tasks that can be completely set-up before $d_i^*$ (line 23). Finally, with the objective of speeding-up the execution of the scheduling algorithm, an array (tried) is used to keep track of the ready tasks that have been unsuccessfully tried to be placed (lines 4, 18 and 21).

The worst-case complexity of FAEDF algorithm is $O(R \cdot (R + E))$ when the “look ahead” is enabled, and $O(R)$ if it is not enabled. In any case, as previously explained, FAEDF includes several ways to effectively reduce the time overheads, e.g., discard the ready tasks which are larger than the largest free rectangle on FPGA early or use the aforementioned tried array.

Finally, we note that our FAEDF algorithm does not consider the overheads introduced when making the scheduling and allocation decisions. However, these should be considered in a real-world application, e.g., a guard increase of the ICAP access time should be allowed.

5.2 Allocation Algorithms

As presented in chapter 3, several research efforts can be found in the technical literature to improve the computation density when allocating hardware tasks onto an FPGA device. These approaches mainly include bin packing-based algorithms (e.g., [Bazargan et al., 2000, Walder et al., 2003]) and adjacency-based heuristics (e.g., [Tabero et al., 2004, Tabero et al., 2006, Marconi et al., 2010]). However, bin-packing algorithms do not consider the effect that placing a task into the “bin box” involves in allocating future tasks and, adjacency-based heuristics have a vision of only one resource row/column beyond the boundaries of already placed tasks.

To tackle these limitations, the novel Empty Area/Volume Compaction heuristics (EAC and EVC) and, as a natural improvement to them, the Snake task allocation strategy are proposed in this thesis.
input: (1) List of R ready tasks, sorted by increasing $d_i^* - t_{ICAP,i}$
(2) List of E executing tasks, sorted by increasing $t_{SP,i} + t_{ICAP,i} + t_{E,i}$
(3) MER, given by the allocator
(4) Real-time deadline tightness, $\sum_{k \in Ready\ Queue} \frac{t_{ICAP,k}}{H_c - t}$
(5) current time $t$

output: Scheduled Task

1. Reset tried array (set all positions equal to false);
2. $i \leftarrow$ First Task in Ready Queue;
3. while $i \leq R$ do
   4. if $MER \geq h_{x,i} \cdot h_{y,i}$ and $\text{tried}[i] = false$ then
      5. if $\text{Allocate}(i) \neq \emptyset$ then
         6. return $i$;
   7. if $\sum_{k \in Ready\ Queue} \frac{t_{ICAP,k}}{H_c - t} < \text{Threshold}$ then
      8. $j \leftarrow$ First Task in Executing Queue;
      9. $F \leftarrow false$;
     10. while $j \leq E$ and $t_{SP,j} + t_{ICAP,j} + t_{E,j} \leq d_i^* - t_{ICAP,i}$ and $!F$ do
         11. if $h_{x,j} \geq h_{x,i}$ and $h_{y,j} \geq h_{y,i}$ then
             12. $F \leftarrow true$;
          13. $j \leftarrow$ Next Task in Executing Queue;
      14. if $F$ then
          15. $m \leftarrow$ Next Task in Ready Queue after $i$;
          16. while $m \leq R$ do
             17. if $t + t_{ICAP,m} \leq d_i^* - t_{ICAP,i}$ then
                 18. if $MER \geq h_{x,m} \cdot h_{y,m}$ and $\text{tried}[m] = false$ then
                     19. if $\text{Allocate}(m) \neq \emptyset$ then
                         20. return $m$;
                     else
                         21. $\text{tried}[m] \leftarrow true$;
                         22. $m \leftarrow$ Next Task in Ready Queue;
                 23. return $\emptyset$;
          24. $i \leftarrow$ Next Task in Ready Queue;
          25. return $\emptyset$;

Algorithm 5.2: Schedule_FAEDF()
Before outlining these, it is important to note that all of the allocation algorithms presented in this section, including the ones we are proposing, manage the FPGA at a very high-level of abstraction. This is only possible because the FPGA area is kept void of static routes and other implementation-related obstacles at all times, as R3TOS is able to do. Indeed, the FPGA is modelled as a grid, named \texttt{FPGA\_state}, where each position represents an FPGA resource or a set of resources. Due to the existing reconfiguration granularity in current Xilinx partially reconfigurable FPGAs (see chapter 2), all of the resources included within the same column of a clock region are mapped to the same position in the grid, i.e., the vertical granularity must be a minimum number of clock regions. On the other hand, the horizontal granularity can be arbitrarily chosen based on the required efficiency in the use of system resources and admissible computational burden. The finest granularity, the best achievable exploitation of FPGA resources, i.e., the exact number of resources required by the tasks can be assigned to them, but with the highest computational burden, i.e., there are more allocations to be evaluated.

### 5.2.1 Empty Area / Volume Compaction Heuristics (EAC / EVC)

EAC and EVC heuristics are aimed at preserving the MER intact for future use as long as possible, trying to place small tasks in the smallest pieces of empty area where they fit, including the areas between damaged resources. Therefore, these heuristics are suitable to be used in R3TOS. In fact, in presence of faults it is not true that the best allocation for a task is always a vertex of a previously allocated task, as adjacency-based heuristics assume (e.g., 2DA or 3DA).

Another benefit of EAC and EVC heuristics is their ability to manage the FPGA device as a single resource instead of splitting it into non-realistic independent pieces of area, as bin packing-based algorithms do (e.g., KAMER). Indeed, these heuristics thoroughly analyse the state of the whole reconfigurable area of the FPGA, giving rise to an Empty Area Descriptor (EAD), which is later consulted when a new task comes. By using the pre-computed information included in the EAD, unfeasible allocations can be discarded early and the quality of the feasible allocations can be quickly determined. As a result, more placement candidates can be evaluated in less time, and eventually, better results can be achieved.
The main difference between EAC and EVC is that the latter also analyses the time domain to prevent future fragmentation, and to achieve higher computation densities.

Algorithms 5.3 to 5.7 show the most important pseudocode portions to compute the EAC and EVC heuristics, while Algorithm 5.8 shows the pseudocode to make the allocation decisions based on these heuristics. The example depicted in Figure 5.6 is used to illustrate the computation of EAC/EVC, with all the related calculations shown in Figure 5.7.

**EAC/EVC: Horizontal Analysis**

The area of the FPGA is firstly analysed in the horizontal direction, from right to left and from left to right (see Algorithm 5.3). Every time the resources associated to the cell in the next column of the FPGA_state grid are available to be used, a counter (named length in Algorithm 5.3) is incremented (line 5) and, in case the resources are not available, the counter is reset (lines 7 and 8). It is assumed that a resource is not available when it is already assigned to another executing task or when it is damaged, but it is considered available when assigned to a task in allocated state\(^1\). Likewise, active data traces (i.e those which are still required by other consumer tasks) are also represented by means of not available (BRAM) cells in the FPGA_state grid. Hence, the set of the counter values at each grid position make up the Right / Left Adjacency Matrices (RAM and LAM) and represent the amount of adjacent free resources in each direction (right or left).

**EAC/EVC: Vertical Analysis**

In a second phase, the vertical analysis is carried out (see Algorithm 5.4). The objective of this analysis is to find the greatest empty rectangle that can be formed at each position in the up-right, up-left, down-right and down-left directions, and results in four new matrices: Up-Right Adjacency Matrix (URAM), Up-Left Adjacency Matrix (ULAM), Down-Right Adjacency Matrix (DLAM), and Down-Left Adjacency Matrix (ULAM). These rectangles are depicted in Figure 5.4. The vertical analysis is based on the geometrical meaning of the RAM / LAM matrices. Indeed, the area of the widest empty rectangle that can be formed at a given position is equal to the product of the RAM / LAM value for that position (named width in Algorithm 5.4).

\(^1\)The decision about which out of the resources assigned to the tasks in allocated state should be re-assigned to other incoming tasks is an important issue that is currently not directly considered by our allocation heuristics. However, this should be specifically considered in a similar way that cache replacement algorithms do in a software OS [Aho et al., 1971].

155
Chapter 5 - R3TOS Algorithms to Manage Workload and FPGA Resources

**Algorithm 5.3: EAC heuristic: Compute_RAM() and Compute_LAM()**

```plaintext
input: FPGA_state (i.e., for each FPGA position Available or Not Available)
output: RAM and LAM

1 for
   i = 0...H_x − 1
   i = H_x − 1...0
   do
      length ← 1;
      for j = 0...H_y − 1 do
         if FPGA_state[i][j] is Available then
            length ← length + 1;
            RAM[i][j] ← length; /* when computing RAM or */
            LAM[i][j] ← length; /* when computing LAM */
         else
            length ← 1;
            RAM[i][j] ← 0; /* when computing RAM or */
            LAM[i][j] ← 0; /* when computing LAM */
      done
   done
```

**Figure 5.4:** Greatest empty rectangles in up-right, up-left, down-right and down-left directions
5.4) and the number of consecutive neighbour locations in the up or down directions with the same or greater \( \text{RAM} / \text{LAM} \) value. Analogously, the area of the highest empty rectangle that can be formed at each position is equal to the product of the number of consecutive neighbour locations in the up or down directions, with a non-zero \( \text{RAM} / \text{LAM} \) value, and the lowest among these values. In general, the area of all empty rectangles that can be formed at each position can be iteratively computed as the product of the successive decrements of the \( \text{RAM} / \text{LAM} \) value for that position (until 0) and the number of consecutive neighbour locations in the up or down directions, with the same or greater \( \text{RAM} / \text{LAM} \) value. Note that this multiplication is computed by the repeated addition of the actual width value in line 9 of Algorithm 5.4. In this algorithm the value of width is decremented in line 13 and the \( k \) index is used to go through the up or down directions. The area of the greatest empty rectangle at each position and in each direction is finally written in the \( \text{URAM} \), \( \text{ULAM} \), \( \text{DRAM} \) and \( \text{DLAM} \) matrices (lines 11, 12 and 14).

For instance, in Figure 5.7, the value of \( \text{URAM}(5,9) \) corresponds to the area of 4x7 (widest) empty rectangle which can be formed in up-right direction. Note that this is the only rectangle that can be formed at that position. On the other hand, the value of \( \text{URAM}(5,2) \) is equal to the area of 3x4 (highest) empty rectangle that can be formed in that direction. Note that at that position the widest rectangle that can be formed is 1x7. Finally, the value of \( \text{URAM}(7,6) \) is equal to the area of either 5x3 or 3x5 empty rectangles. At that position the highest empty rectangle that can be formed is 7x2.

**EAC: Area Adjacency Analysis**

As shown in Algorithm 5.5, the third and last phase of EAC heuristic computation consists in adding the aforementioned four matrices \( \text{URAM}, \text{ULAM}, \text{DRAM} \) and \( \text{DLAM} \) to give rise to the 2-D Adjacency Matrix (2DAM). Conceptually, the values stored in each position of 2DAM represent in what measure that position contributes to form adjacent pieces of empty area.

**EVC: Time and Area Adjacency Analysis**

As previously mentioned, EVC extends the area analysis to include the time domain. Although it is inspired by the 3DA heuristic, some changes are introduced with the objective of reducing the computational burden when making the allocation decisions (see Algorithm
Chapter 5- R3TOS Algorithms to Manage Workload and FPGA Resources

Algorithm 5.4: EAC heuristic: Compute_URAM(), Compute_ULAM(), Compute_DRAM() and Compute_DLAM()

input : RAM (for URAM and DRAM computation) and LAM (for ULAM and DLAM computation)
output: URAM, ULAM, DRAM and DLAM

for $i = 0...H_x - 1$ do
  for $j = 0...H_y - 1$ do
    \[
    \text{width} \leftarrow \text{RAM}[x][y]; \quad \text{/* when computing URAM / DRAM or */}
    \]
    \[
    \text{width} \leftarrow \text{LAM}[x][y]; \quad \text{/* when computing ULAM / DLAM */}
    \]
    \[
    \text{area}_{\text{max}} \leftarrow \text{width};
    \]
    \[
    k \leftarrow j;
    \]
    while width > 0 do
      \[
      \text{area} \leftarrow 0;
      \]
      while $k < H_y$ and $k \geq 0$ and
        \[
        \text{RAM}[i][k] \geq \text{width} \quad \text{/* when computing URAM / DRAM or */}
        \]
        \[
        \text{LAM}[i][k] \geq \text{width} \quad \text{/* when computing ULAM / DLAM */}
        \]
        do
          \[
          \text{area} \leftarrow \text{area} + \text{width};
          \]
          \[
          k \leftarrow k + 1; \quad \text{/* when computing URAM / ULAM or */}
          \]
          \[
          k \leftarrow k - 1; \quad \text{/* when computing DRAM / DLAM */}
          \]
          if area > area_{\text{max}} then
            \[
            \text{area}_{\text{max}} \leftarrow \text{area};
            \]
            \[
            \text{width} \leftarrow \text{width} - 1;
            \]
          enddo
        enddo
      enddo
    enddo
  enddo
enddo

Algorithm 5.5: EAC heuristic: Compute_2DAM()

input : URAM, ULAM, DRAM and DLAM
output: 2DAM

for $i = 0...H_x - 1$ do
  for $j = 0...H_y - 1$ do
    \[
    \text{2DAM} \leftarrow \text{ULAM}[i][j] + \text{URAM}[i][j] + \text{DLAM}[i][j] + \text{DRAM}[i][j];
    \]
  enddo
enddo

Algorithm 5.5: EAC heuristic: Compute_2DAM()
In order to create a task-independent set of data which could be used at runtime for any coming task, a time window $T_W$ equal to the greatest execution time in the task-set is chosen ($T_W = \max\{t_{E,i}\}$). For instance, in the example shown in Figure 5.6, $T_W = \max\{5, 8, 6\} = 8$.

For each position, the temporal adjacency within the time window with device's boundaries (lines 9, 15, 21 and 27), with other executing tasks (lines 8, 14, 20 and 26), and with damaged resources in the four directions (lines 6, 12, 18 and 24) is computed. As a result, the Temporal Adjacency Matrix ($TAM$) is obtained. The value stored in each position of the $TAM$ represents in what measure that position contributes to increasing the computation density. More specifically, a high temporal adjacency value means that the adjacent FPGA resources will remain occupied for a long time, while a low temporal adjacency value means that the adjacent resources will be released soon. For instance, in Figure 5.7, the top-left position has a temporal adjacency with device's boundaries equal to $8 + 0 + 0 + 8 = 16$; the same as for position (10,2), but for the latter the adjacency is with a damaged resource and an executing task.

The temporal adjacency information ($TAM$) is then combined with the area adjacency information ($2DAM$) to create the 3-D Adjacency Matrix ($3DAM$), as shown in Algorithm 5.7. The operation used to combine both time and area domains is the division (see line 3). Therefore, when the $2DAM$ value is high (i.e., that location is part of a great adjacent free area) and the $TAM$ value is low (i.e., that location does not contribute to keeping the tasks compacted in the Computation Cube), the resulting $3DAM$ value is very high (i.e., disadvantageous); and, on the other hand, when the $2DAM$ value is low and the $TAM$ value is high, the resulting $3DAM$ value is very low (i.e., advantageous). For the rest of the cases, the resulting $3DAM$ value is medium.

**The Empty Area Descriptor (EAD)**

As the tasks are placed relative to their upper-left vertex (see Figure 5.6a), the aforementioned $DRAM$ and $RAM$ matrices are especially useful to describe the state of the FPGA reconfigurable area. Consequently, these two matrices are included in the EAD. Each value stored in the $DRAM$ matrix indicates the biggest empty rectangle available in the bottom-right direction. Therefore, a task can be placed in a given position only if its area is less than or equal to the actual $DRAM$ value stored at that position. To account for shape aspects, the
input: FPGA_state and the state of the tasks
output: TAM

for $i = 0...H_x - 1$ do
  for $j = 0...H_y - 1$ do
    TAM[$i$][$j$] $\leftarrow$ 1;
    // Bottom
    if $j - 1 \geq 0$ then
      if FPGA_state[$i$][$j-1$] is Damaged then
        TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+$T_W$;
      else if Task @ ($i$,$j-1$) is Executing then
        TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+Remaining $t_e$ of Task @ ($i$,$j-1$);
    else
      TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+$T_W$;
    // Left
    if $i - 1 \geq 0$ then
      if FPGA_state[$i-1$][$j$] is Damaged then
        TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+$T_W$;
      else if Task @ ($i-1$,$j$) is Executing then
        TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+Remaining $t_e$ of Task @ ($i-1$,$j$);
    else
      TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+$T_W$;
    // Top
    if $j + 1 < H_y$ then
      if FPGA_state[$i$][$j+1$] is Damaged then
        TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+$T_W$;
      else if Task @ ($i$,$j+1$) is Executing then
        TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+Remaining $t_e$ of Task @ ($i$,$j+1$);
    else
      TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+$T_W$;
    // Right
    if $i + 1 < H_x$ then
      if FPGA_state[$i+1$][$j$] is Damaged then
        TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+$T_W$;
      else if Task @ ($i+1$,$j$) is Executing then
        TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+Remaining $t_e$ of Task @ ($i+1$,$j$);
    else
      TAM[$i$][$j$] $\leftarrow$ TAM[$i$][$j$]+$T_W$;

Algorithm 5.6: EVC heuristic: Compute_TAM()
Chapter 5- R3TOS Algorithms to Manage Workload and FPGA Resources

input : URAM, ULAM, DRAM, DLAM and TAM
output: 3DAM

1. for $i = 0 \ldots H_x - 1$
   2. for $j = 0 \ldots H_y - 1$
   3. \(3DAM \leftarrow 2DAM[i][j]/(TAM[i][j] + 1)\);

Algorithm 5.7: EVC heuristic: Compute_3DAM()

RAM matrix is used. A task can be placed in a given position only if its width is less than or equal to the actual RAM value stored at that position. In order to accelerate the search of feasible allocations for the tasks, the highest value in each column of the DRAM (named column_MER) is also saved in the EAD. column_MERs permit to discard all the positions of a column without having to individually analyse each of these positions. Note that depending on the geometry of the targeted FPGA, it may be preferable to save the set of row_MERs instead of column_MERs. The EAD includes the MER as well, which is equal to the maximum value in the DRAM. As previously introduced, the MER is given to the scheduler in order to discard unfeasible to place tasks early, when making the scheduling decisions.

The last component of the EAD is the 2DAM (when using the EAC heuristic) or 3DAM (when using EVC). As previously explained in this chapter, the values contained in these matrices represent the importance of each position to keep empty area compact and thus, they are used to evaluate the quality of the feasible allocations, as explained in section 3.

The hierarchical structure of the EAD is shown in Figure 5.5.

EAC/EVC-based Allocation Decision Making

At runtime, when a new task comes, the set of feasible allocations for it are evaluated based on the pre-computed values stored in the 2DAM, when using the EAC heuristic, or in the 3DAM, when using the EVC heuristic. As shown in Algorithm 5.8, either an EAC or an EVC score is assigned to each feasible allocation \((x, y)\) of a task \(\theta_i\). Note that unfeasible allocations are discarded early based on the EAD (see lines 4 and 6). The EAC and EVC scores are computed as the sum of the 2DAM or 3DAM values corresponding to the resources to be assigned to \(\theta_i\) in the allocation being evaluated (line 10). The placement quality is inversely proportional to the EAC and EVC scores. Conceptually, a low score means that the adjacent empty area in the device is not significantly fragmented when allocating the task in that po-
Figure 5.5: EAD structure
sition. In the case of EVC, a low score also ensures a good compactness of the tasks within
the Computation Cube (CC). Therefore, the final placement decision consists in selecting
the feasible allocation with the lowest EAC or EVC score (lines 11, 12 and 13). Note that
this way of functioning allows for dealing with different task shapes (i.e., non-rectangular
tasks\(^2\)).

The benefit of EAC and EVC when coping with permanent damage is illustrated in Fig-
ure 5.6. According to both 2DA and 3DA heuristics, and considering the adjacency with the
damage as well, \(\theta_i\) would have been allocated at the bottom-left vertex of the FPGA (can-
didate B) with a 2DA score equal to 14 and a 3DA score equal to 74. The 2DA score for the
candidate A is only 10 and the 3DA score for this candidate is only 56. On the other hand, the
lowest EAC and EVC scores are obtained for candidate A; i.e., EAC = 947 and EVC = 614. The
scores for candidate B are: EAC = 65 \cdot 20 = 1300 and EVC = 623. Therefore, according to both
EAC and EVC heuristics \(\theta_i\) is placed at candidate allocation A. Hence, as shown in Figure
5.6, both 2DA and 3DA heuristics lead to a reduction of the MER from 48 (when placing the task
at position A using either EAC or EVC) to 30, making it more difficult to allocate greater area
tasks coming in the future.

By using the pre-computed 2DAM and 3DAM matrices, the evaluation of each placement
candidate for a task \(\theta_i\) can be done very quickly, involving only \(h_{x,i} \cdot h_{y,i}\) additions. Fur-
thermore, note that the required time for making the allocation decisions of large and small
tasks tends to balance: while there are more feasible candidate allocations for a small task
rather than for a large task, the quality of each candidate is evaluated faster for small tasks
as the number of additions to be done is lower.

On the other hand, the EAD updating is a time-consuming process which is performed
in parallel with the setting-up of the last allocated task in order to improve system perfor-
ance. Indeed, as the scheduling algorithm is not preemptive, the next scheduling point
will not be before the task is completely set-up in the device. The 2DAM and 3DAM matrices
are updated with the area state expected by then: the resources assigned to the task being
set-up are marked as not available, and the resources assigned to the executing tasks which
are expected to finish by then are marked as available.

\(^2\)Despite implementing non-rectangular partially reconfigurable tasks is not currently fully supported by
Xilinx designs tools, in some applications this might be interesting.
Chapter 5 - R3TOS Algorithms to Manage Workload and FPGA Resources

Algorithm 5.8: Allocation selection based on the EAC and EVC heuristics: Allocate_EAC() and Allocate_EVC()
Chapter 5- R3TOS Algorithms to Manage Workload and FPGA Resources

(a) FPGA area state \( t_{e,i} \) is the remaining execution time of task \( i \)

(b) Candidate A (MER\(_A\) = 48)

(c) Candidate B (MER\(_B\) = 30)

Figure 5.6: Allocating \( \theta_i \) at candidate positions A and B

Overall, the worst-case complexity of EAD updating is \( O(C \cdot H_x \cdot H_y + 4 \sum_{i=1}^{H_x} \sum_{j=1}^{H_y} i \cdot j) \approx O(H_x^2 \cdot H_y^2) \), where \( C = 3 \) when using the EAC heuristic and \( C = 4 \) when using the EVC heuristic. For both heuristics the runtime allocation decision making has a worst-case complexity of \( O(h_{x,i} \cdot h_{y,i} \cdot (H_x - h_{x,i} + 1) \cdot (H_y - h_{y,i} + 1)) \). Hence, unlike most of allocation algorithms, whose complexity depends on the number of allocated tasks, the complexity of EAC and EVC heuristics depends on the size of the FPGA. The benefit comes from the fact that not feasible candidates can be discarded early by consulting the EAD, significantly reducing the effective amount of time needed to make the allocation decisions.

With the amount of time available to update the EAD limited by the shortest ICAP access time of the tasks (i.e., \( \min(t_{ICAP,i}) \)), an effective way to speed-up this process is to increase the used granularity at the expenses of losing efficiency in the management of FPGA resources. Note that efficiency is of outmost importance when using small FPGAs, where EAD updating time is not so critical, but it is less important when using large FPGAs which involve longer updating times. Based on this, and also arguing that the type of computation necessary to update the EAD is suitable to be accelerated by hardware (e.g., Algorithms 5.3 to 5.7 have regular data dependencies and, as shown in Figure 5.7, the URAM, ULAM, DRAM and DLAM matrices can be concurrently computed), we posit that the amount of time needed
### Figure 5.7: TIME ANALYSIS (EVC)

**Phase 1:** Horizontal Adjacency & Temporal Adjacency

**Phase 2:** Vertical Adjacency

**Phase 3:** 2-D / 3-D Adjacency Matrix Computation

### AREA ANALYSIS (EAC)

- **Area Analysis (EAC)**: Shows the distribution and analysis of areas in the circuit, helping to optimize resource allocation and improve performance.

---

**Note:** The diagram illustrates the flow of data and the computation process in the FPGA situation shown in Figure 5.6.
to complete the updating can be kept within reasonable bounds, enabling the use of the proposed heuristics in future FPGA devices, with presumably faster reconfiguration speed and more logic resources. A hardware implementation of an EAD updater is described in chapter 6.

### 5.2.2 Snake Task Allocation Strategy

While EAC and EVC reduce the negative effect provoked by external fragmentation, they do not directly consider some key aspects in RC, such as inter-task communications (i.e., hardware tasks are assumed to be independent), usable clock frequency and FPGA resource heterogeneity. Hence, the allocation decisions may result in low performance due to intensive use of ICAP to exchange data among tasks or due to the fact tasks are not executed at their highest clock rate. To tackle these issues we propose the *Snake* allocation strategy \cite{Iturbe2011d}.

Besides promoting the reuse of previously configured circuitry, *Snake* also tries to reuse the intermediate partial results between different computation stages when dealing with non-critical High Bandwidth Communication (HBC) tasks. Note that when a task is non-critical a single instance of it is executed on the FPGA and there is no need to check the correctness of its results. On the other hand, as explained in chapter 4, the results computed by critical tasks must be accessed through the ICAP in order to check their correctness using the CRC32 module included in the HWuK. Hence, EAC and EVC heuristics continue to be useful for allocating triplicated critical tasks and non-critical Low Bandwidth Communication (LBC) tasks. That is, non-critical HBC tasks (which need a long time to exchange data through the ICAP) are allocated with the objective of reducing the ICAP occupation, at the expense of increasing external fragmentation on the device, and LBC tasks (which need short time to exchange data through the ICAP) are efficiently allocated on the resulting FPGA area fragments.

With the objective of speeding-up the computation, *Snake* tries to execute each task at its highest allowed clock frequency\(^3\), especially LBC tasks that require longer time to be completed. However, this must be carefully treated to avoid allocation problems due to the fact that each FPGA clock region can allocate a maximum of two tasks running at different clock rates.

\(^3\)As BUFR_DIVIDE parameters on BUFRs allows values ranging between 1 and 8, there can be only up to 8 different clock rates in the device to be used by the tasks.
clock frequencies; i.e., there are only two regional clock nets to distribute the clock signals in a clock region. In light of making up large regions with the same clock domain where future (large) tasks could be allocated, it is preferable to allocate the tasks with similar clock rates together in the same or adjacent rows and the tasks with radically different clock frequencies in separate rows (see Figure 5.8).

Despite it would be interesting to adjust the clock rate of the tasks during their execution phase (i.e., feed them with a slower clock frequency to allocate in the same rows other tasks which can only run at that clock rate), this capability is not yet included in R3TOS.

![Figure 5.8: Allocation of tasks running at different clock rates (FPGA is rotated 90°)](image)

Summing up, while reusing circuitry and partial results speeds-up the set-up phase of the tasks (better use of ICAP, i.e., time), an optimal management of clocking resources accelerates their execution phase (better use of FPGA resources, i.e., area). However, usually it is impossible to simultaneously take advantage of both improvements. When the execution time of a task is significantly longer than its set-up time (i.e., LBC tasks), it is preferable to feed the task with the highest clock rate despite this results in longer ICAP occupation. On the other hand, when the set-up time of a task is in the same range of its execution time (i.e., HBC tasks), circuitry and/or data reuse is promoted. Indeed, note that HBC tasks usually complete their computation within a relatively short amount of time and hence, the occupation of the clocking resources is not a major problem.

In order to increase the allocatability of the tasks that include more scarce BRAM-based data buffers (typically HBC tasks), several versions of the same task are to be provided. As shown in Figure 5.9, each of the task versions uses different IDB and ODB locations and de-
Chapter 5- R3TOS Algorithms to Manage Workload and FPGA Resources

Figure 5.9: Different implementation versions of the hardware tasks

finishes a different direction of computation (i.e., data flow from the IDB to the ODB). By using the appropriate task version at each time, the task can leave its results in the easiest accessible BRAM memories to be accessed by the subsequent data consumer tasks. In Figure 5.9, the white arrows represent the computation direction, which is vertical for task versions a to d and horizontal for versions e to i. Given the granularity of FPGA’s configuration memory (see chapter 2) and aiming at best exploiting FPGA resources, the tasks must always span a minimum number of clock regions in height. Therefore, typically the size of the IDB and ODB will be an integer multiple of 4 BRAMs (72 Kb). However, for efficiency reasons, both IDB and ODB could be mapped to the same BRAM column; i.e., each buffer using 2 BRAMs. In the horizontal direction the criterion changes. Tasks with horizontal computation direction must lie between columns of BRAMs and the width of tasks with vertical computation direction is chosen with the only constraint of fitting the necessary amount of resources. This means that a pair a-c or a-d or b-c or b-d task versions can flexibly exploit the FPGA resources between two BRAM columns. We acknowledge a memory requirement increase to store the bitstreams associated to each version of the tasks. However, the memory overhead is admissible considering the benefit this method allows. Otherwise, Snake is very limited.

Figure 5.10 shows the allocation decision-making diagram of Snake. When a task $\theta_i$ is scheduled, Snake checks first whether $\theta_i$ is critical or non-critical, and HBC or LBC. If $\theta_i$ is non-critical and HBC, Snake checks whether it is preferable to reuse circuitry (i.e., circumvent $t_{A,i}$), or reuse partial results (i.e., circumvent $t_{R,j}$ and $t_{D,i}$, where $\theta_j$ is the data producer task of $\theta_i$). Note that while circuitry can be reused only if the task remains still configured
New Scheduled Task $\theta_i$?

BRAM-based buffers?

Use EAC/EVC Heuristics

Evaluate Sharing the ODB

is $t_{A,i} > t_{R,j} + t_{D,i}$?

yes

no

is $\theta_i$ allocated?

yes

no

Scheduler

Critical

Non-critical

LBC

HBC or LBC

HBC

LBC

Critical

Non-critical

EAC/EVC used to select the most appropriate location

Deliver highest clock frequency

$t_{R,i}$ Reduction

Re-use Circuitry

$t_{R,i}$ Circumvention

EAC/EVC Heuristics

No Performance Enhancement / Minimal Fragmentation

Re-use Circuitry

$t_{R,i}$ Circumvention

Evaluate (Sharing the ODB)

Re-use Partial Results

$t_{R,i}$ Partial Results

$t_{R,i}$ & $t_{D,i}$ Circumvention

EAC/EVC used to select the most appropriate DRT

Use DRT

$t_{R,i}$ & $t_{D,i}$ Reduction

Data reuse

Producer's ODB

Whole FPGA

Snake's Allocation Evaluation Order

HBC

Critical

Less fragmentation (EAC/EVC Heuristics)

Non-critical

Shortest ICAP occupation (Partial results reuse)

LBC

Critical

Shortest FPGA resources occupation (Execute at highest clock frequency)

Non-critical

Figure 5.10: Snake task allocation strategy
Figure 5.11: Reusing partial results with *Snake*
on the FPGA, partial results can always be potentially reused as the data traces of HBC tasks are stored in BRAMs. Reusing an already configured task is immediate as no allocation decisions must be made, i.e., the task is simply executed on the same position where it was last time. However, when reusing the partial results, some allocation decisions are to be made. The best case is when \( \theta_i \) can directly access its input data from \( \theta_j \)’s ODB (i.e., \( \theta_j \)’s ODB is used as \( \theta_i \)’s IDB), but this requires there are sufficiently large amounts of contiguous resources to allocate \( \theta_i \) next to \( \theta_j \)’s ODB. Preferably, consumer tasks are allocated opposite to producer tasks, to keep the latter allocated on the FPGA, promoting future circuitry reuse. If strictly necessary, producer tasks are deallocated and their resources assigned to consumer tasks. If there are several versions of \( \theta_i \) which fit in the free area next to \( \theta_j \)’s ODB, EAC/EVC heuristics are used to select the one that minimises the fragmentation on the device. Note that BRAM access switching between producer and consumer tasks can be done very quickly, but requires a resource sharing policy to avoid conflicts as only one of the tasks can access BRAMs at the same time. On the other hand, if there is no sufficiently large free area to allocate any implementation version of \( \theta_i \) next to \( \theta_j \)’s ODB, the feasibility of using a Data Relocation Task (DRT) is evaluated. By using a DRT the set of data can be rapidly moved (through the functional layer) from its current location to a new position where it is accessible by the consumer task. If no DRT can be used, the allocation decisions are made using the EAC/EVC heuristics, and the data is delivered to the consumer task through the configuration layer. Summing up, for non-critical HBC tasks, \textit{Snake} starts evaluating the feasible allocations near the data producer task, continues evaluating the FPGA allocations which are reachable by means of DRTs and finally, it switches to analyse the whole FPGA locations seeking for the lowest EAC/EVC score (i.e., minimal fragmentation).

As shown in Figure 5.11, the linking together of the hardware tasks by means of the memory elements where the data traces are temporarily stored leads to computation chains on the FPGA. This gives \textit{Snake} its name. The task chains are initiated in the main CPU’s ODBs (\textit{Heads}) and the results computed by the last task in the chain are copied through the configuration layer of the FPGA to the main CPU’s IDBs (\textit{Tails}), where they are accessible by the software program. As shown in Figure 5.11b, \textit{Snake} is an efficacious way to deal with the heterogenous resource columns embedded in modern FPGAs as well as to circumvent the damaged resources in the chip.
5.3 Simulation Results

This section presents the obtained results when simulating our scheduling and allocation algorithms. The simulation experiments cover a wide range of task parameters and different damage situations in the chip. Finally, an estimation of the performance improvement brought about by Snake using a realistic heterogeneous FPGA device model is provided.

5.3.1 Simulation Set-Up

A discrete-time simulation framework was built to evaluate the performance of the proposed scheduling and allocating algorithms. The framework ran under Windows XP OS on an Intel Core Duo CPU @ 3 GHz.

The framework simulated a Virtex-4 XC4VLX160 device with up to 12 clock regions, 3 BRAM columns, 1 DSP48 column and a sandbox of 28 CLB columns width. Based on the layout of this FPGA, shown in Figure 5.12a, the vertical granularity was set to be a clock region (i.e., $H_y=12$), while the horizontal granularity was set to be either 4 CLB columns or a single heterogeneous resource column (i.e., $H_x = 15$).

Due to the lack of a common benchmark for RC systems, we resorted to creating our own synthetic hardware tasks. Different task-sets, each containing up to 60 hardware tasks were randomly generated. The execution deadlines, execution times and sizes of the tasks were appropriately chosen, starting from random values, in order to simulate different $U_{ICAP}$ and $U_{COMP}$ situations\(^4\). For the sake of simplicity, the allocation time of the tasks was considered to be equal to their size $t_{A,i} = h_{x,i} \cdot h_{y,i}$. We also considered that successive instances of the tasks were released with the shortest allowed time between them. Under this worst-case assumption, which is similar to the situation considered when proposing the definition of $U_{COMP}$ in chapter 3, aperiodic tasks can be considered periodic.

Up to 10,000 experiments were performed for each $U_{ICAP}$ and $U_{COMP}$ situations and the obtained results were averaged. All tasks were set ready at time 0 (i.e., critical instant) and each experiment was considered to be finished when every task in the task-set had either met or missed its execution deadline at least once. Three different cases have been studied.

\[ U_{ICAP} = \sum \theta_i \cdot t_{ICAP,i} \]  
\[ U_{COMP} = \frac{1}{H_x \cdot H_y} \cdot \sum \theta_i \cdot \frac{(t_{ICAP,i} + t_{E,i}) \cdot h_{x,i} \cdot h_{y,i}}{D_i} \]  

\(^4\)As presented in chapter 3, $U_{ICAP} = \sum \theta_i \cdot t_{ICAP,i}$ and $U_{COMP} = \frac{1}{H_x \cdot H_y} \cdot \sum \theta_i \cdot \frac{(t_{ICAP,i} + t_{E,i}) \cdot h_{x,i} \cdot h_{y,i}}{D_i}$.
Figure 5.12: Left half part of the simulated XC4VLX160 part
In cases A and B, we did not consider inter-task dependencies, inter-task communication overheads or resource heterogeneity, i.e., the tasks could be arbitrarily allocated on the sandbox of the FPGA. Furthermore, all tasks were considered to run at the same clock frequency. We note that this is the most commonly simulated scenario in related work. Since our scheduling and allocation algorithms are soft real-time, non-preemptive and designed for 2-D area model, they were only compared with equivalent non-preemptive EDF scheduling, working with 2DA/3DA allocation heuristics. Indeed, EDF is one the most consolidated soft real-time scheduling algorithms and adjacency based heuristics show the best allocation results in the current state of the art, being used, or serving as inspiration, in some of the latest research efforts in the field (see chapter 3). In order to complete the characterisation of our EAC/EVC heuristics, in case B, up to 25 CLBs within the sandbox were marked as damaged (approximately 0.5% of the total CLBs in the sandbox). For fair comparison, we provided 2DA and 3DA heuristics with a mechanism for dealing with faults, namely the damaged CLB positions were added as additional vertexes in the Virtex List Set (VLS).

Four metrics were used to evaluate the performance of our algorithms:

1. **Missed Deadlines (MD):** is the percentage of missed execution deadlines.

2. **Scheduling Feasibility (SF):** refers to the percentage of feasible schedules produced; i.e., percentage of schedules that do not miss any deadline.

3. **Exploited Computation Volume (ECV):** refers to the use of the CC by the hardware tasks which meet their deadlines to perform active computation. Note that the set-up phase of the tasks is not considered when computing the ECV.

4. **Algorithm's Execution Time (AET):** refers to the amount of time needed for making the scheduling and allocation decisions per executed task, as well as the time needed for updating the EAD.

In case C, all of the previously neglected RC-related issues were included in the simulation to evaluate our realistic Snake task allocation strategy. Hence, this simulation considered the whole FPGA device, i.e., sandbox and heterogeneous resource columns. Inter-task dependencies were randomly generated, with a maximum of 3 dependencies per task, and the amount of time needed to exchange data among tasks was also considered; i.e., $t_{Di}$ and
The data delivery/retrieval time was uniformly distributed in [90%.110%] of the execution time for HBC tasks and in [30%.50%] for LBC tasks. HBC tasks and LBC tasks were randomly generated with a similar proportion of BRAM to CLB columns in the device; i.e., 15 to 1 more LBCs. All of the tasks were considered not critical. It was assumed that four implementation versions were available for each task, with vertical and horizontal computation direction, and for each computation direction with the IDB and ODB located in reverse positions. The data buffers of HBC tasks were considered to be implemented using 4 BRAMs. When a consumer task accessed data directly from its producer task’s ODB, an acceleration factor of 6x was assumed in inter-task communications. Likewise, when DRTs could be used, it was assumed an acceleration factor of 1.5x. Furthermore, clocking aspects were envisaged: the execution time of the tasks depended on the used clock frequency, and the amount of tasks running at different clock frequencies in a row was limited to two. The highest clock frequency at which each task could run was randomly selected, ranging from 1x (i.e., base clock rate) to 5x.

5.3.2 Case A: No Damage in the Device

Figure 5.13 shows the collected results in three representative situations with no damaged resources in the simulated FPGA device: (1) when the FPGA resources are highly utilised \(U_{COMP} = 0.9\) and the real-time constraints are tight \(U_{ICAP} = 0.9\), (2) when the FPGA resources are highly utilised and the real-time constraints are moderate \(U_{ICAP} = 0.75\), and (3) when the FPGA utilisation is medium \(U_{COMP} = 0.75\) and the real-time constraints are tight \(U_{COMP} = 0.9\). Note that the results shown in this figure are normalised to the highest value.

As expected, the results are better when either time aspects were considered when making the allocation decisions or when area aspects were considered when making the scheduling decisions; i.e., EVC outperforms EAC and FAEDF outperforms EDF. The improvement is more noticeable when the extra dimension was considered only once; i.e., the benefit of including time aspects when making the allocation decisions is greater with EDF, which does not account for area aspects, than with FAEDF, which already considers area aspects. In all of the simulated situations FAEDF-EVC shows the best results, namely, less amount of missed deadlines, higher rate of feasible schedules, and better exploitation of computation volume,
Figure 5.13: Performance with no damage in the FPGA device (Note: $EAT(U)$ refers to the EAD updating time, $EAT(A)$ refers to the time needed for making the allocation decisions and $EAT(S)$ is the time needed for making the scheduling decisions)
while EDF-2DA shows the worst results. Moreover, the experiments conducted confirm that FAEDF-EAC produces slightly better results than EDF-3DA, despite the former only considers the area domain.

Including area aspects when making scheduling decisions FAEDF results in approximately double execution time. Nonetheless, this penalty is admissible as the scheduling decisions can always be made in less than 8 microseconds. On the other hand, the time overhead introduced by EAC and EVC heuristics derived from the process of updating the EAD is more significant, around 100 microseconds. This might seem excessive considering the high clock speed used in the simulations (3 GHz), which is definitely not available for FPGAs. However, two aspects must be considered here. First, the conducted simulation experiments do not account for neither the acceleration brought about by custom hardware implementation of the algorithms nor parallelism in the area matrices computation. Second, the EAD updating time is to be reduced when the FPGA area is split into three independent Fault Containment Computation Regions (FCCRs), as explained in chapter 4. Contrasting with the long time required to update the EAD, we note the remarkable achievable acceleration when using our allocation heuristics to make the allocation decisions. Indeed, these were made in around 35 microseconds on average when using 2DA/3DA heuristics and in less than 10 microseconds when using our EAC/EVC heuristics (around 71% speed-up).

Summing up, when using FAEDF-EAC/EVC, the scheduling and allocation decisions can be made online in less than 20 microseconds (this time is approximately double when using EDF-2DA/3DA), but there is an overhead due to EAD updating process, which is in the range of 100 microseconds in our simulation framework running at 3 GHz. Therefore, the efficacy of our algorithms highly depends on the success in speeding-up the EAD updating process. Anyway, we point out that some overhead is still admissible as the EAD updating can be parallelised with task set-up phase.

5.3.3 Case B: Damage in the Device

Figure 5.14 to Figure 5.17 show the measured performance for various FPGA utilisation situations and different real-time constraints in the presence of permanent damage on the FPGA device. As can be seen in the figures, most of the performance metrics (e.g., MD, SF and ECV) show an exponential variation with the number of simulated faults in the FPGA.
The most important conclusion obtained from these results is the capability of EAC/EVC heuristics to deal with permanent damage in the FPGA. For instance, unlike 2DA/3DA, EAC/EVC heuristics are able to produce feasible schedules (i.e., no missed deadlines) for all of the simulated fault situations when the FPGA utilisation is low ($U_{COMP} = 0.5$, see Figures 5.14 and 5.16). Despite the fact it is not possible to produce feasible schedules in the rest of the cases, the differences between the results obtained by both heuristics are still appreciable. Namely, when using EAC/EVC, between 5% and 20% fewer deadlines are missed and a similar improvement is measured in the exploitation of FPGA's computation volume. The difference is greater when the FPGA is highly utilised, e.g., 20% improvement when $U_{COMP} = 0.9$ (see Figure 5.17) and only 5% improvement when $U_{COMP} = 0.75$ (see Figure 5.15). In addition, it is interesting to check that EAC/EVC heuristics show better results with regard to 2DA/3DA as the FPGA gets more damaged for most of the range of simulated faults. Finally, note that the performance of EVC and EAC is similar, with the former producing slightly better results.

Unlike MD, SF and ECV, the execution time of the allocation algorithms, AET, shows a nearly linear increase with the number of simulated faults when using EAC/EVC heuristics. Indeed, the time needed to update the EAD in the worst situation is measured around 160 microseconds. Again, although the amount of time needed to make the scheduling decisions is always greater when using FAEDF than when using EDF, it is admissible (i.e., less than 3 microseconds). On the other hand, the time needed to allocate the tasks when using 2DA/3DA heuristics increases exponentially with the number of damaged resources in the chip, reaching up to 250 microseconds when the FPGA is highly utilised and significantly damaged (see Figure 5.17). Notably, this is even longer than the time needed to update the EAD in that situation. Hence, it cannot be claimed the online allocation capability for the 2DA/3DA heuristics when dealing with partially damaged FPGAs. On the other hand, under the same conditions, the time needed to make the allocation decisions when using EAC/EVC heuristics is only 24 microseconds, which is an admissible overhead to target online task allocation. This important improvement is the result of the capability to discard unfeasible to allocate tasks early by both the scheduler (based on the MER size) and by the allocator (based on the column_MERs in the EAD). Finally, as expected, the execution time of EVC is slightly longer than that of EAC.
Figure 5.14: Simulation results when low utilisation of the FPGA and loose real-time constraints: $U_{ICAP}=0.5$ and $U_{COMP}=0.5$

Figure 5.15: Simulation results when medium utilisation of the FPGA and moderate real-time requirements: $U_{ICAP}=0.75$ and $U_{COMP}=0.75$
Figure 5.16: Simulation results when low utilisation of the FPGA and tight real-time requirements: $U_{ICAP}=0.9$ and $U_{COMP}=0.5$

Figure 5.17: Simulation results when high utilisation of the FPGA and loose real-time requirements: $U_{ICAP}=0.5$ and $U_{COMP}=0.9$
5.3.4 **Case C: Snake Task Allocation Strategy**

Figure 5.19 shows the results obtained when simulating the *Snake* approach on the realistic RC scenario described in section 5.3.1. The results are normalised to the highest value and in all of the cases FAEDF scheduling algorithm was used. We note that the fact of simulating most of the RC issues results in lower performance when using FAEDF-EAC and FAEDF-EVC than shown in previous simulations. For instance, the ECV was significantly smaller and most of the produced schedules were unfeasible.

Based on the obtained results, we conclude that *Snake* improves the performance shown when exclusively using EAC/EVC heuristics. This is reasonable as it is indeed especially conceived to extend these heuristics to deal with the simulated RC particularities and issues in this experiment (e.g., inter-task dependencies and communications).

An important aspect to note is that the average time spent when making the allocation decisions in *Snake* is considerably reduced, as there is no need to evaluate all of the feasible allocations on the FPGA. When the tasks are reused, no allocation decisions must be made, and when the partial results are reused, only one allocation must be evaluated (for each version of the task). Moreover, when DRTs are used only a few more allocations need to be evaluated, namely those where DRTs are able to move input data from producer's ODB. In our simulations a maximum of 26 target allocations were considered when using DRTs: up to 4 clock regions above and below the ODB where the data is held, in the same BRAM column as well as in the neighbour right and left columns (see Figure 5.18).

5.4 **Chapter Conclusion**

In this chapter, a novel real-time scheduling algorithm (FAEDF), two novel allocation heuristics (EAC/EVC), and a novel allocation strategy (*Snake*) were proposed. Notably, these approaches consider most of the particularities and limitations derived from the use of current partially reconfigurable FPGAs for computing purposes, and have been shown to be efficacious by means of synthetic simulations. First, the FAEDF scheduling algorithm improves non-preemptive EDF by delaying the execution of tasks which cannot be allocated on a first instance until enough adjacent free area is released on the FPGA. Second, the EAC/EVC heuristics help to reduce the fragmentation on the device, which is especially useful when
the FPGA is partially damaged, and increase the computation density, which is essential in some modern lightweight applications (e.g., [Ma et al., 2013]). Significantly, the computations required by EAC/EVC heuristics are to be carried out while the tasks are configured through the ICAP, thus allowing considerably fast online allocation decisions to be made. Finally, the *Snake* task allocation strategy tries to remedy a weaknesses of R3TOS, namely the intensive use of the ICAP.

Based on the simulations, the EAC heuristic is chosen to be used in R3TOS as it is simpler to implement in hardware than EVC and still produces good quality results, which indeed are better than those produced by 3DA even if this also considers time aspects. A preliminary implementation of a scheduler and allocator using our algorithms is reported in [Hong et al., 2011], and a thorough description of them is given in chapter 6. Having said this, we acknowledge the differences that might well exist between the simulated scenario and any real-world scenario.
Figure 5.19: Performance on a realistic RC simulation scenario when using the Snake approach (Note: \textit{EAT(U)} refers to the EAD updating time, \textit{EAT(A)} refers to the time needed for making the allocation decisions and \textit{EAT(S)} is the time needed for making the scheduling decisions)
Low-Level Hardware Support for the
R3TOS HWuK

The R3TOS HWuK includes specific circuitry to speed-up the making of the scheduling and allocation decisions (i.e., to execute the algorithms presented in chapter 5) as well as to control the OS mechanisms which run in the background, some of them concurrently. Most of this circuitry is located within the specific region assigned to HWuK on the FPGA (e.g., the scheduler, allocator and configuration manager), but some parts are distributed along the FPGA device (e.g., the logic to manage the clocking resources). While all this circuitry is static and operates uninterruptedly, the HWuK also includes some dynamic circuitry which is loaded when required to command the execution of the computing tasks, i.e., Task Control Logic (TCL), and to give support for communications, i.e., Data Relocation Tasks (DRTs).

Although the general architecture of HWuK has been outlined in chapter 4, this chapter describes in detail the internal structure and functioning of each HWuK component and the interactions among them. Table 6.1 shows an itemisation of the main set of functionalities implemented by each of the HWuK components, while the interconnections of the latter components are depicted in Figure 4.19 (chapter 4).

While I have planned the architecture and functioning of the HWuK as well as designed most of the circuitry described in this chapter, I acknowledge the support received from

Part of this chapter has been accepted for publication in [Iturbe et al., 2013a].
my colleagues in the System Level Integration Research Group, Mr Chuan Hong and Mr Ali Ebrahim, in implementing some specific parts. Mr Hong coded the software routines executed by the PicoBlazes included in the scheduler and allocator [Hong et al., 2011], i.e., mainly the routines which execute the scheduling and allocating algorithms, and Mr Ebrahim developed the core functionality of the FSM included in the ICAP driver [Ebrahim et al., 2012], i.e., the states which directly interact with FPGA’s configuration logic.

We report a minimalist implementation of the HWuK in the current stage of development. Despite the fact that our current implementation is able to schedule, allocate, configure and execute hardware tasks on the FPGA, the complete integration has not been done yet.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Scheduler</th>
<th>Allocator</th>
<th>Conf. Man.</th>
<th>TCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SCHEDULING:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling of hardware tasks</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ALLOCATION &amp; DEALLOCATION:</strong></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Allocation of hardware tasks</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deallocation of hardware tasks</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>INTER-TASK COMMUNICATIONS:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Management of data traces</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ICAP-based data communication</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRT-based data communication</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>SYNCHRONISATION:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poll HWS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Take HWS</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Give HWS</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>MISCELLANEOUS FUNCTIONING:</strong></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Management of clocking resources</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Timing of task phases</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Interaction with the main CPU</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>RELIABILITY:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tripling hardware task instances</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Voting computed results</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>FAULT-HANDLING:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scrubbing of soft-errors</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Detection of damaged resources</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

*Table 6.1: Mapping of R3TOS functionality to main HWuK components*
6.1 R3TOS Static Infrastructure

The R3TOS static infrastructure gives support for the execution of the hardware tasks. It basically includes the R3TOS core (i.e., HWuK and main CPU), and the clock-tree which delivers the clock signal along the chip (i.e., BUFRs and BUFGCTRLs). In addition, R3TOS static infrastructure includes the instantiation of the specific device primitives required by the HWuK (i.e., STARTUP, ICAP and Frame_ECC logic).

Unlike in related work, the regional clocking resources are not dedicated to any specific hardware task, but shared among all of the tasks placed in the same clock region. By doing so, tasks can be clocked using multiple BUFRs located in different clock regions and thus, their maximum height is not constrained. Moreover, not including the BUFRs in the architecture of the tasks enhances their (horizontal) allocatability.

The clock-tree is managed based on performance and reliability premises. Indeed, the clock-tree is a single-point of failure and must be carefully hardened to increase the reliability of the system [Kopetz, 2011]. In the initial configuration, the two BUFRs located in the rightmost/leftmost columns of each clock region are directly connected to the two regional clock nets, but these connections can be changed at runtime to recover from failed clock sources. Besides, the BUFR_DIVIDE parameter of the BUFRs are individually configured on-the-fly to set-up the required clock rate for the hardware tasks they feed at each time.

To detect damaged regional clocking resources, a diagnostic circuit is implemented in each clock region, next to the BUFRs. As shown in Figure 6.1 this circuit is very simple, fitting in only 8 slices and thus spanning only one CLB column. It takes two clock sources as inputs: the global clock signal which is input to the BUFR to diagnose and the regional clock signal which is output. The global clock feeds a delay line composed of 5 cascaded latches which capture the regional clock signal at rising edges, and another latch which capture the regional clock signal at falling edge. As the frequency of the regional clock signal can be equal or up to 8 times slower than that of the global clock, the values stored in all of the latches can never be the same. Note that when their frequencies are the same, the values captured at falling and rising edges must be different, and when their frequencies are different, the 5 delay line latches store values corresponding to more than a half period of the regional clock signal and hence, at least one of them must be different. Therefore, the error signal
must be always '0'. If this is not met, then the regional clock signal is stuck at the same logic level, i.e., the BUFR is broken. In order to make the diagnostic result remotely accessible from the HWuK without using static routes across the chip, the error signal is registered in a RAM-LUT, whose content can be read-back using the ICAP. Once the latter RAM-LUT is accessed, the HWuK is responsible for clearing it back to '0'.

Up to three BUFGCTRLs are used in R3TOS. One of them feeds the BUFRs (BUFR clk), with its rate being very high to allow a wide range of different smaller frequencies to be generated to drive the tasks (e.g., 250, 166, 125, 100, 83, 71 and 62 MHz can be derived from a raw clock rate of 500 MHz). The remaining two BUFGCTRLs are used to feed the main CPU and the HWuK, respectively. While the clock that feeds the HWuK is always running, the clock that feeds the main CPU and the clock that feeds the hardware tasks must be both stopped prior to performing an access to either any of the BRAM memories on the chip or to the STARTUP primitive. This is done to prevent corrupting the data stored in the BRAMs, in case either the main CPU or the hardware tasks use dual-ported BRAMs, and to ensure safe access to internal signals in the configuration logic of the FPGA.

For reliability purposes, each input of the BUFGCTRL, I0 and I1, is connected to an independent clock source, clk_i0 and clk_i1, respectively. The clock selection port of the BUFGCTRL is driven by the Dual-clock Management Circuit (DMC) depicted in Figure 6.2. This circuit takes both clock sources, clk_i0 and clk_i1, as inputs. Specifically, clk_i0 is used as the clock in the circuit and clk_i1 is captured in two latches, each working at falling and rising edges. Assuming both input clock signals are of the same frequency, the values in the latches can never be the same, except when any either clk_i0 or clk_i1 do not work; i.e., they are stuck-at the same logic level. Hence, the XOR between the values stored in the latches (sel_OK1) is used to select the clock source in the BUFGCTRL.
automatically switching to clk_i0 when clk_i1 fails. In order to circumvent problems when the two clock sources are not synchronised, BUFGCTRL clock selection port is driven by sel_OK1 signal delayed, ensuring there is sufficient guard time between the last clock pulse generated by the active clock source before it gets damaged, and the clock pulse immediately after, which is generated by the switched clock source.

We note that errors due to phase and frequency deviations in the clock sources are not directly handled by the DMC circuit, but they can be detected by DCMs via their LOCKED port.

Despite the DMC circuit is able to recover from a single failed clock source, it is convenient that the R3TOS HWuK is aware of the latter situation with the objective of preventing potential system failures in case the remaining clock source fails (see chapter 7). Therefore, an error signal is generated by XORing sel_OK1 and sel_OK2 signals, where the latter is derived from a diagnostic circuit which operates in the contrary way to that which generates sel_OK1 signal; i.e., clk_i1 is used as the clock in the circuit and clk_i0 is captured in the two latches. Both error and sel_OK1 signals are delivered to the R3TOS HWuK to enable it keeping track of the state of the clock sources.

![Schematic](Figure 6.2: Dual-clock Management Circuit (DMC))
As presented in chapter 2 and 3, static routing is a critical issue when building an RC system as the relocated hardware tasks may use the routing resources already assigned to static routes in the target position. To limit the static routing in our design, the R3TOS core and the clocking diagnostic circuits are constrained within Partially Reconfigurable Regions (PRRs). This is an effective way to keep the rest of the chip free of static routes. As shown in Figure 6.3, the only signals which extend beyond the boundaries of the PRRs are the clock lines, which indeed do not constrain the allocatability of the tasks as they are separately managed by R3TOS. Since Xilinx design tools do not allow to include neither IOBs nor STARTUP, ICAP and Frame_ECC primitives in the PRR assigned to R3TOS, these components are connected to the R3TOS circuitry through Bus Macros (BMs).

6.2 Task Control Logic (TCL)

So far we have seen that TCLs act as proxies for the HWuK wherever the tasks are placed within the FPGA, i.e., the execution of a hardware task is remotely enabled by asserting its Hardware Semaphore (HWS) in the TCL. Furthermore, TCLs provide a means for carrying out data exchanges among the tasks; i.e., data is copied from producer task’s Input Data Buffer (IDB) to consumer task’s Output Data Buffer (ODB). In short, TCLs wrap the hardware tasks to provide a common interface for them.

The implementation of a TCL depends on the type of task it is to be attached to. For instance, large data buffers, typically associated with High Bandwidth Communication (HBC) tasks, are implemented using high-density yet more scarce and location-specific FPGA storage resources (namely BRAMs), while small data buffers, typically associated with Low Bandwidth Communication (LBC) tasks, are implemented using low-density and more abundant storage resources (namely LUTs). Table 6.2 shows the most important generic parameters to be specified when synthesising TCLs.

6.2.1 Functioning: Data-stream processing vs. Hardware Acceleration

While the TCL of a data-stream processing task is suited to handle a set of Processing Elements (PEs) which operate in a pipeline fashion, the TCL of a hardware accelerator task is designed to deal with random accesses to different positions in its data buffers.
Figure 6.3: R3TOS static infrastructure
### Table 6.2: Generic parameters in the TCL

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_width</td>
<td>Word-length in the IDB</td>
</tr>
<tr>
<td>results_width</td>
<td>Word-length in the ODB</td>
</tr>
<tr>
<td>n_16_data</td>
<td>Used with LUT-based data buffers: ⌈# of input data/16⌉</td>
</tr>
<tr>
<td>n_16_results</td>
<td>Used with LUT-based data buffers: ⌈# of output results/16⌉</td>
</tr>
<tr>
<td>n_bram_data</td>
<td># of BRAMs used in the IDB</td>
</tr>
<tr>
<td>n_bram_results</td>
<td># of BRAMs used in the ODB</td>
</tr>
<tr>
<td>N</td>
<td>Pipeline depth in data-stream processing tasks</td>
</tr>
</tbody>
</table>

As shown in Figure 6.4a, in a TCL oriented for data-stream processing tasks, both wr_addr and wr_en signals are equal to rd_addr and rd_en signals, respectively, albeit delayed by N clock cycles. N is one of the generic parameters to be explicitly configured by the user in the HDL code at design time and it is equal to the depth of the pipeline. Since access to data is sequential, a counter is used to generate the addresses. In this scenario, rd_en and wr_en signals are permanently active while input data is being processed (i.e., HWS=’1’), and turn inactive when the last position of the ODB is written (e.g., when all wr_addr bits are ’1’). Note that this type of TCL is compatible with most of the systems generated with currently available design tools for signal processing. For instance, a pipelined signal processing system built with Xilinx System Generator includes as entity ports: (1) clock, (2) inputs and outputs, to be connected to TCL’s IDB and ODB, respectively, and (3) clock enable signal, to be directly connected to TCL’s HWS [Xilinx Inc., 2008a].

On the other hand, as shown in Figure 6.4b, the implementation of the TCL of a hardware accelerator task is much more open, conceptually limited to the connection of data, address and control ports of the PEs to the data buffers.

In Figure 6.4a and 6.4b, the solid-line arrows represent the logical access through the ICAP to the information stored in the data buffers, and the broken-line arrows refer to the physical access through the functional layer to (only) BRAM-based data buffers. The latter access is performed either by DRTs or data consumer tasks when reusing partial results.

The HWS is the same for all of the types of tasks and it is implemented on a single LUT-RAM as shown in Figure 6.5. Some remarks are in order here. First, in order to ensure that both configuration layer (HWuK) and functional layer (hardware task) access the same bit within the LUT-RAM, the lock_pins attribute is used. Second, in order to circumvent...
Chapter 6- Low-Level Hardware Support for the R3TOS HWuK

data corruption due to simultaneous access from configuration and functional layers, all of
the inputs of the LUT-RAM, i.e., D, A(3..0), WE and WCLK, are kept at logic level ‘0’ except
when the task_end pulse is triggered. In order to do so, both the WCLK and WE ports
of the LUT-RAM are fed by the same task_end pulse. To guarantee that the logic value
on the WE port is stable by when the pulse reaches the WCLK port, a clock cycle delay is
forced in the latter port by routing the task_end signal through a latch in its way to it. In
connection with this, special care must be taken to ensure that writable LUTs (i.e., LUT-RAM
or SRL16) are not mapped to the same CLB column where the HWS is located. Otherwise,
their content could be corrupted when polling the HWS.

The HWS acts as local reset for the registers of a particular hardware task. Note that a
good practice for FPGA design is to use a reset input in all of the registers to initialise them
with a known state at startup. The startup state of the registers is coded in the SRMODE
bits in the configuration bitstream. However, hardware description languages also allow for
initialising the registered signals to specific values without using any reset; e.g., in VHDL,
the sig_reg signal is initialised at logic level ‘1’ as

\[
\text{signal sig_reg: std_logic := '1'}. \]

In the latter case, the specified initialisation values are coded in the INIT bits in
the configuration bitstream. These values are loaded in the actual registers by issuing a
GRESTORE command when the configuration bitstream is transferred to the FPGA's con-
figuration memory. By using the STARTUP primitive, and conveniently using the masking
possibility of flip-flops, R3TOS is able to perform a local GRESTORE operation exclusively
affecting the registers contained in a particular hardware task \(^1\). Therefore, by combining
GCAPTURE and HWS it is indeed possible to fully exploit the capabilities delivered by mod-
ern HDLs.

6.2.2 Implementation: LUT-based Data Buffers vs. BRAM-based Data Buffers

LUT-based data buffers are used when dealing with low amounts of data; e.g., coeffi-
cients of a filter or extracted features from large data-sets. Specifically, as shown in Figure
6.6, non-writable LUT4 primitives are used to implement IDBs and writable LUT-RAM prim-
itives are used to implement ODBs. Since LUT4s can be mapped to SLICEMs and LUT-RAMs

\(^1\)We note that Xilinx Inc. has recently made available a specific constraint, RESET AFTER RECONFIG, to
automatically initialise the registers of a reconfigurable area after reconfiguring it, thus circumventing the ne-
cessity of issuing GRESTORE command [Xilinx Inc., 2012b].
Chapter 6 - Low-Level Hardware Support for the R3TOS HWuK

... a hardware-accelerator task

... a data-stream processing task

Figure 6.4: Simplified generic architecture of TCL for ...

attribute lock_pins : string;
attribute lock_pins of HWS : label is "ALL";

HWS : RAM16X1S
generic map (INIT => X"0000")
port map (O => task_start,
A0 => '0',
A1 => '0',
A2 => '0',
A3 => '0',
D => '0',
WCLK => task_end after 0 ps,
WE => task_end)

Figure 6.5: HWS implementation
Chapter 6: Low-Level Hardware Support for the R3TOS HWuK

![Figure 6.6: LUT-based data buffers](image)

(a) ODB VHDL code

```vhdl
LUT_ODB : for i in 0 to n_16_results-1 generate
begin
  LUT_ODB_bits : for j in 0 to results_width-1 generate
  attribute lock_pins : string;
  attribute lock_pins of RAM16X1S_odb : label is "ALL";
  begin
    RAM16X1S_odb : RAM16X1S
    generic map (
      INIT => X"0000")
    port map (
      O => unconnected_output(i)(j),
      A0 => wr_addr(0),
      A1 => wr_addr(1),
      A2 => wr_addr(2),
      A3 => wr_addr(3),
      D => results(i)(j),
      WCLK => comp_clk,
      WE => wr_en(i)
    );
  end generate LUT_ODB_bits;
end generate LUT_ODB;
```

(b) IDB VHDL code

```vhdl
LUT_IDB : for i in 0 to n_16_data-1 generate
begin
  LUT_IDB_bits : for j in 0 to data_width-1 generate
  attribute lock_pins : string;
  attribute lock_pins of LUT4_idb : label is "ALL";
  begin
    LUT4_idb : LUT4
    generic map (
      INIT => X"0000")
    port map (
      O => data_mux(i)(j),
      I0 => rd_addr(0),
      I1 => rd_addr(1),
      I2 => rd_addr(2),
      I3 => rd_addr(3)
    );
  end generate LUT_IDB_bits;
end generate LUT_IDB;
```

(c) LUT-based IDB implementation (data_width=32, n_16_data=1)
must be mapped to SLICEs, both IDB and ODB can be implemented using a single CLB column. Indeed, LUT-based buffers are more amenable to optimisation than BRAM-based ones as the storage granularity is smaller, i.e., only 16 bits in Virtex-4. Regarding Figure 6.6, four implementation remarks are in order. First, the pins of both LUT4 and LUT-RAM primitives must be locked in order to ensure data integrity when copying the data from the LUT-RAM of a producer task’s ODB to the LUT4 of a consumer task’s IDB (i.e., \texttt{lock\_pins} attributes are used). Second, \texttt{S} attributes are used to prevent the trimming of LUT-RAMs during the synthesis. Note that the output port of these elements remains unconnected as they are accessed only through the ICAP. Third, an address decoding circuitry is needed to enable the appropriate LUT-RAM where the output results are to be saved, and an input multiplexer is needed for delivering the appropriate input data to the tasks’ PEs at each time. The latter multiplexer, which is included in the IDB’s glue logic, takes as inputs \texttt{data\_mux(i)} signals. Finally, in order to enable high data rates (e.g., an input data reading / output data writing per clock cycle), each bit of the same data is mapped to a different LUT. Otherwise the data should be read bit by bit and stored in registers for direct access, taking 16 clock cycles to do so.

Figure 6.7 shows the resource requirements for implementing the LUT-based data buffers as a function of their sizes. In this figure, note the excessive area overhead when the data sizes are significantly large.

Indeed, when dealing with large amounts of data, BRAM-based buffers are preferred with the port width of the BRAMs being determined by the bit-length of the data to be pro-

![Figure 6.7: Resource requirement for different LUT-based buffers size](image-url)
BRAM_ODB : for i in 0 to n_bram-1 generate
begin
  RAMB16_odb : RAMB16_S9_S36
  port map (
    DOA => open,
    DOB => open,
    DOPA => open,
    DOPB => open,
    ADDR_A => wr_addr,
    ADDR_B => (others => '0'),
    CLKA => comp_clk,
    CLKB => '0',
    DIA => results,
    DIB => (others => '0'),
    DIPA => (others => '0'),
    DIPB => (others => '0'),
    ENA => wr_en(i),
    ENS => '0',
    ENSR => '0',
    WEA => '1',
    WEB => '0'
  );
end generate BRAM_ODB;

(a) ODB VHDL code (data_width=32)

BRAM_IDB : for i in 0 to n_bram-1 generate
begin
  RAMB16_idb : RAMB16_S9_S36
  port map (
    DOA => data_mux(i),
    DOB => open,
    DOPA => open,
    DOPB => open,
    ADDR_A => rd_addr,
    ADDR_B => (others => '0'),
    CLKA => comp_clk,
    CLKB => '0',
    DIA => (others => '0'),
    DIB => (others => '0'),
    DIPA => (others => '0'),
    DIPB => (others => '0'),
    ENA => rd_en(i),
    ENS => '0',
    ENSR => '0',
    WEA => '0',
    WEB => '0'
  );
end generate BRAM_IDB;

(b) IDB VHDL code

(c) BRAM-based ODB implementation

Figure 6.8: BRAM-based data buffers
Chapter 6 - Low-Level Hardware Support for the R3TOS HWuK

cessed by the task’s PEs (e.g., 8, 16 or 32 bits). As shown in Figure 6.8, data buffers are distributed along the four BRAMs in the same FPGA column to make the best use of BRAM content frames. By doing this, 78% of the frame bits can be used as data, and this value can reach up to 87.8% when the BRAM parity bits are also used as data. Going into implementation details, \( \text{rd_addr}(\log_2(\frac{256 \cdot n_{\text{bram data}}}{\text{data width}}))^{\downarrow} \) bits are used in the IDB multiplexer to select the appropriate \( \text{data_mux}(i) \) to be delivered to the task's PEs and, \( \text{wr_addr}(\log_2(\frac{256 \cdot n_{\text{bram results}}}{\text{results width}}))^{\downarrow} \) bits are used in the ODB to generate the appropriate \( \text{wr_en} \) signals for the BRAMs. The remaining address bits are directly connected to the BRAMs.

Figure 6.9 shows the resource requirements for implementing BRAM-based data buffers with different sizes and word-lengths.

![Figure 6.9: Resource requirement for different BRAM-based buffer configurations](image)

Two functioning remarks are in order here. First, when dealing with tasks which consume data from multiple producers, each producer is assigned a different data segment in the consumer’s IDB. Furthermore, with the objective of improving performance, each data segment spans a multiple of 256 bits within the BRAMs, i.e., data segments are mapped to different frames. Second, a typical way to deal with tasks that produce a variable amount of data is to use a LUT to keep track of the amount of valid data stored in the tasks’ BRAMs, and use this information to determine the exact number of frames that need to be copied to consumer’s ODB. The LUT is necessary as the data in the BRAM content frames is stored in a non-trivial way (see chapter 2).
6.3 Data Relocation Tasks (DRTs)

DRTs include all the necessary circuitry to move data from a source BRAM-based ODB to a target BRAM-based IDB through the functional layer of the FPGA. This includes both the logic to drive the BRAMs and the wires to connect them. DRTs are only used with HBC tasks.

The way that a DRT is attached to data producer and consumer tasks is depicted in Figure 6.10a. The stripped regions represent the routing of the BRAMs, which is dynamically changed to permit access to the memories either from the computing tasks or from the DRT. As shown in this figure, DRTs temporarily modify the internal architecture of data producer task’s TCL, namely the interconnections of its ODB’s BRAMs. The clock signal delivered to the BRAM column is stopped prior to carrying out the latter modification, thus preventing corruption of the data stored in the memories due to unexpected early activity while reconfiguring BRAM interconnections. Note that the BRAMs are the left and right boundaries for both DRTs and computing tasks, but they extend in opposite directions. Therefore, the BRAM ports are connected to the FPGA resources located in opposite sides in DRTs and computing tasks, ensuring no routing conflicts occur when switching the BRAM interconnections, i.e., the Programmable Interconnection Points (PIPs) which are active in computing tasks are not active in DRTs and vice versa\(^2\) (see Figure 6.11).

Access to the target BRAMs from the consumer task is automatically gained when configuring its bitstream, i.e., BRAM interconnection frames are overwritten (See Figure 6.10b). Since the consumer task is configured while the DRT is transferring data, in order to ensure the integrity of the data in the transfer process it must be ensured that \(t_{A,\text{consumer}} > t_{E,\text{DRT}}\), where \(t_{E,\text{DRT}}\) is the amount of time needed by the DRT to transfer all data. In order to leave the FPGA in the same state as it was before using the DRT, thus preventing future routing conflicts, two steps must be made: (1) the DRT must be deallocated by blanking all the necessary configuration frames and, (2) the temporarily modified data producer task’s TCL must be restored by rewriting the original BRAM interconnection configuration.

In order to start communications from the HWuK, a HWS-like LUT-RAM is included in DRTs, named Data Relocation Enable (DRE). The DRE is enabled through the ICAP only

\(^{2}\)Both DRTs and computing tasks are assigned different partially reconfigurable regions and synthesised following the Xilinx partial reconfiguration flow.
Figure 6.10: DRTs

(b) Management of DRTs: Timing diagram

when the source and target BRAMs are correctly connected by means of the DRT. Then, data is sequentially read from producer task's ODB (source BRAM) and copied, one clock cycle after, to consumer task's IDB (target BRAM). This process is repeated until the last data is copied. Afterwards, the DRE is automatically disabled by DRT's own logic. Note that this functioning is similar to data-stream processing tasks, but without transforming data.

As shown in Figure 6.12, the logic to drive the BRAMs is mapped to a single CLB column, namely to the CLB column located next to the source BRAMs. The rest of the DRT is free of logic, it only includes the wires to connect the source and target BRAMs together. In order to reduce the amount of wires, the width of the BRAM port is set to 8-bit when using DRTs. Furthermore, an input multiplexer is used to select the data delivered by only one of the source BRAMs at any time, i.e., the DRT receives $8 \cdot n\_bram$ input data wires, which are time-multiplexed using only 8 wires. Hence, the total amount of wires to be routed to the target BRAMs is: $19+n\_bram$, where $n\_bram$ are the BRAM enables, 11 are address and 8 are data. Using this scheme and working at 100 MHz, it is possible to transfer the content of up to 4 BRAMs within less than a hundred of microseconds. In any case, higher amounts of
Figure 6.11: Representation of the BRAM switching between a computing task and a DRT
Chapter 6- Low-Level Hardware Support for the R3TOS HWuK

Figure 6.12: DRT implementation ($h_x = 1$ and $h_y = 4$)
data can be transferred within the same time when the frequency of \( \text{clk}_{\text{comms}} \) is higher. We note that \( \text{clk}_{\text{comms}} \) is set-up in the same way as the clock delivered to any computing task. In general: 

\[
E_{DRT} = n_{\text{bram}} \cdot 2^{11} \cdot \frac{1}{f_{\text{clk}_{\text{comms}}}}.
\]

The benefit of the above implementation is that the configuration information of the DRT is mainly included in the 20 interconnection frames associated to source and target BRAMs as well as in the 22 frames of the CLB column where the logic is implemented. The remaining configuration information of the DRTs consists of a set of active PIPs, which can be grouped into a reduced number of frames by exploiting the regularity of the FPGA routing structure (see Figure 6.12), i.e., same PIPs in different Switch Matrices (SMs) within the same FPGA column are mapped to the same frames, with the ultimate objective of reducing the amount of time needed to allocate the DRTs, \( t_{A,DRT} \).

DRTs are managed as if they were standard computing tasks, where their height and width are equal to the vertical and horizontal distance between the source and target BRAMs they connect. As standard computing tasks, they can be used only when the region where they are to be allocated is completely free.

The lack of flexibility of using DRTs to communicate tasks when compared with on-line routing is to be compensated by providing a wide range of differently sized and shaped DRTs to cover plenty of miscellaneous source-target BRAM interconnections (e.g., BRAMs located in the same column, two columns apart, etc). Furthermore, it could be useful to connect multiple sets of source BRAMs to a single set of target BRAMs to deal with tasks which consume data from multiple producers. In this case, the input multiplexer and target BRAM enable logic should be adjusted in the DRT to copy the data of each producer task to the appropriate data segment in the consumer’s IDB.

Figure 6.13 shows the requirements imposed by DRTs in terms of occupied slices and amount of frames necessary to be configured. Note that only DRTs which connect BRAMs located in adjacent columns are shown in this figure (i.e., \( h_x = 1 \)). Also note that the number of active PIPs do not significantly increase as the (vertical) distance between the BRAMs to connect increases (i.e., as \( h_y \) increases). This is the result of three factors: (1) use only 8 multiplexed data wires, (2) exploit the regularity of FPGA’s routing structure, (3) use of hex lines when the BRAMs to connect are located in different rows. On the other hand, the number
Chapter 6- Low-Level Hardware Support for the R3TOS HWuK

of active PIPs and frames inevitably increases when using a larger amount of data buffers (i.e., more BRAMs) as the DRT receives more input data signals and also uses more slices to implement the input multiplexer and the rest of its internal logic. Based on this figure, DRTs make it possible to accelerate data transfers among BRAMs by an approximate factor of: \(\frac{128}{85}\) (1.5x when \(n\_bram=4\)), \(\frac{256}{112}\) (2.3x when \(n\_bram=8\)) and \(\frac{512}{179}\) (2.8x when \(n\_bram=16\)). However, the key benefit of DRTs is the possibility to create large computation chains on the FPGA with the capability to speed-up the computation when they are reused to process different data.

![Figure 6.13: Resource and configuration requirement for different DRTs (\(h_x = 1\))](image)

### 6.4 Scheduler

Chapter 5 described the Finishing-Aware EDF (FAEDF) scheduling algorithm in terms of conceptual scope. This section, on the other hand, focuses on the practical implementation issues of FAEDF using real FPGA hardware. Besides, it covers other functionality assigned to the scheduler (e.g., management of the state of the tasks and interaction with the main CPU).

All this functionality is implemented by a Xilinx PicoBlaze processor and the related information is stored in a Task BRAM. In order to keep track of time, the scheduler receives an interruption from a kernel timer at every kernel tick \(t_{KT}\), which is in the range of tens-hundreds of microseconds. The latter timer is also responsible for checking that the scheduler is working as expected (i.e., it does not get stuck).
6.4.1 Task BRAM

Figure 6.14 shows the internal organisation of the Task BRAM. This memory can hold up to 60 different hardware tasks, and for each task, a data segment of 32 8-bit words is reserved to store the task’s information. Using 8-bit memory words ensures the best area-speed trade-off as it matches with the PicoBlaze’s bus width. The task data segments are located at predefined positions within the memory (i.e., task with TaskID $i$ spans positions $32 \cdot i$ to $32 \cdot i + 31$). This allows for immediate access to the information of a given task.

The information stored of each hardware task $\theta_i$ in its data segment is the following:

- **Implementation dependent parameters**, including data buffer type (BRAM or LUT based), and for each version of the task: width $h_{x,i}$, height $h_{y,i}$, (4) allocation time $t_{A,i}$, highest usable clock frequency (i.e., BUFR\_DIVIDE), internal architecture descriptor and internal relative location of the HWS, IDB and ODB. Currently, there is space for only 2 versions of each task, with vertical and horizontal direction of computation. As the IDB and ODB can be located at different positions in each of the versions, when both data buffers are implemented using the same technology (i.e., BRAM or LUT), it is possible to deal with tasks which move data from right to left (R2L), from left to right (L2R), from top to bottom (T2B) and from bottom to top (B2T).

- **Functioning dependent parameters**, including data delivery time $t_{D,i}$, worst-case execution time $t_{E,i}$ (i.e., when using the slowest clock rate), data retrieval time $t_{R,i}$, relative execution deadline $D_i$, data producer task ID, number of data consumer tasks, a criticality bit to indicate whether the task needs to be triplicated or not, and a bit to indicate whether the results computed by the task are required by the main CPU. Note that all of these parameters can be changed by the main CPU each time the task is triggered based on the user specifications.

- **Some specific parameters to be maintained by the HWuK**, including $(x, y)$ allocation of the three redundant instances of the task and position of their data traces within the Fault Containment Computation Regions (FCCRs), number of times the data trace has been accessed, state flags, etc. State flags specify which version of the task is in use, whether the voting process has already been performed on the computed results.
Figure 6.14: Organisation of the Task BRAM
by the task, or whether the task is a standard hardware task or an exclusively communicating task used to model hardware-software interactions (see chapter 4). Note that the absolute position of any IDB, ODB or HWS within the FPGA can be easily known by adding its relative internal location within the task to the position \((x, y)\) where the task is placed at any time.

The tasks are dynamically organised in different queues based on their state at any time (Executing, Ready and Allocated). Hence, the scheduler can perform two kinds of state related consults: (1) retrieve the list of all of the tasks which are in a specific state, by consulting the corresponding state queue, or (2) check in which state is a specific task, by consulting its data segment within the Task BRAM. As required by the FAEDF algorithm, the Ready queue is sorted by increasing \(d^*_i - t_{ICAP_i}\), and the Executing queue is sorted by increasing \(t_{SP_i} + t_{ICAP_i} + t_{E,i}\). On the other hand, the Allocated queue is not sorted. Note that the task being set-up at any time is also treated as a queue with a single element.

Pointers are used to keep the tasks linked in the queues. Each queue has a **Queue-Head_Pointer** which points to the first task in the queue. These pointers are located at pre-defined positions within the Task BRAM: 0x7F0 for the Allocated queue, 0x7F1 for the Setting-up queue, 0x7F2 for the Ready queue and, 0x7F3 for the Executing queue. Likewise, each data segment includes a **Next_Task_Pointer** to point to the next task in the queue. Note that the **Next_Task_Pointer** of the last task in a queue is equal to 0xFF and therefore, the **Queue_Head_Pointers** are equal to this value when their respective queues are empty. Hence, the **Next_Task_Pointer** of the task being set-up is always equal to this value.

In this context, the tasks are accessed sequentially in the queues, starting from the head. Note that this is a very flexible structure which can be dynamically sized, i.e., queues grow and shrink as tasks are added and removed from them. Indeed, a task can be easily added or removed by simply modifying its **Next_Task_Pointer** and the **Next_Task_Pointer** of the previous task in the queue. These operations are performed by the scheduler’s PicoBlaze each time a task changes its state. Task transitions and their implications in the Task BRAM are depicted in Figure 6.15, where the state of this memory is shown as \(\theta_2\) goes through its life cycle. It is assumed that \(\theta_2\) has a tighter deadline than \(\theta_0\) and hence, it is
inserted in the first position of the Ready queue (Figure 6.15b). Also it is assumed that $\theta_2$ is
to be finished before $\theta_1$, but after $\theta_3$. Therefore, it is inserted in-between these tasks in the
Executing queue (Figure 6.15d). On the other hand, tasks are always inserted in the first po-
sition of the Allocated queue, directly pointed by the Queue_Head_Pointer (see Figure
6.15c).

Figure 6.15: Task state transitions (Note: thick lines represent the pointers that must be updated)

The Task BRAM includes 12 pre-defined positions to exchange data with the main CPU,
which accesses them from the other port of the memory. Note that the scheduler accesses
this memory in 8-bit chunks while the main CPU accesses it in 32-bit chunks. Memory posi-
tions 0x7F4 to 0x7FB are used by the HWuK to receive data coming from the main CPU (IN),
and positions 0x7FC to 0x7FF are used to send data to the main CPU (OUT). Hence, there
are 8 IN positions in order to receive the aforementioned 6 application changeable parameters, a corresponding TaskID and a control byte. Every time the most significant byte of IN (b7) is written, an interruption is generated to the scheduler’s PicoBlaze, and when a new output data is written to the most significant byte of OUT (b3), an interruption is generated to the main CPU. As the CPU can only write to IN positions in the Task BRAM, the triggered tasks are passed one by one to HWuK. When this occurs, the latter updates the stored information in the data segment of the triggered tasks and temporarily buffers their TaskIDs in a Ready Buffer, which is located in memory positions 0x782 to 0x791, prior to inserting them in the ready queue. The reason behind delaying the insertion of the tasks is because of the time it takes to find the suitable position in the ready queue for them. Therefore, the Next_Task_Pointer is the only parameter which is not immediately updated in the Task BRAM when a new task is marked as ready by the main CPU.

Other buffers are reserved in the Task BRAM to store the tasks which have missed their deadlines (i.e., Missed Dealsines Buffer), the tasks which are to be deallocated (Deallocated Buffer) as well as the tasks which have finished executing (Finished Buffer) or which are to finish by the next scheduling point (Finishing Expected buffer). Indeed, the tasks which have missed their deadlines or which have completed their computation need to be buffered prior to being passed to the main CPU as the latter only receives tasks one by one. Likewise, the tasks to be deallocated need to be buffered until the ICAP is available to perform the required updating in the configuration memory, and the tasks expected to finish by the next scheduling point are buffered prior to being passed to the allocator to simulate the future FPGA state.

Finally, some memory positions in the Task BRAM are reserved to be used as the PicoBlaze processor’s stack, namely positions 0x7D2 to 0x7EF.

6.4.2 The Scheduler’s PicoBlaze

The scheduler’s PicoBlaze is the central core of the HWuK, mastering its operation. Besides making scheduling decisions and keeping tasks buffered in corresponding state queues, it is responsible for interfacing with the main CPU and for commanding the configuration manager to perform the required operations at any time. Furthermore, despite the fact it relies on the allocator for making allocation decisions, the scheduler’s PicoBlaze implements
some area-related functions as well; e.g., decide which tasks must be deallocated and keep track of the data traces on the FPGA.

One of the most important issues when implementing any scheduling algorithm is how to manage time. The endless time model used in the conceptual scheduling algorithm in chapter 5 must be adapted to a digital time model with a finite horizon. To circumvent the problems related to time counter overflow, R3TOS conveniently keeps track of limited amounts of time (see Figure 6.16). Namely, in order to measure the amount of time that tasks remain in each state, they are assigned a Task_Timer within their data segments which are decremented every $t_{KT}$.

![Figure 6.16: Time management in the R3TOS scheduler](image-url)
When a task $\theta_i$ is released at $r_i$, its Task_Timer is initialised with $D_i - t_{E,i} - t_{ICAP,i}$, for standard tasks, and with $D_i - t_{E,i} - 2 \cdot t_{R,i} - t_{ICAP,i}$, for tasks that need to interact with the main CPU. Note that in both cases $t_{ICAP,i}$ is the worst-case estimated amount of time the task will need to access the ICAP. That is, no performance improvement is considered (e.g., reuse of circuitry or partial results and circumvention of the voting of producer task’s results). To compute $t_{ICAP,i}$, the data retrieval time of the data producer task $t_{R,j}$ is needed, which can be easily accessed at Task BRAM position $32 \cdot j + 2$. If the data producer task is a software task, $t_{R,j}$ is the data retrieval time from the ODB of the main CPU, which is stored at memory position $0x781$; i.e., software tasks are modelled as standard hardware tasks with TaskID=60. Note that the latter parameter is modifiable at runtime depending on the amount of data that needs to be exchanged.

In this context, the updated Task_Timers are equal to $d_i^* - t_{ICAP_i} - t$ when the tasks are in ready state and, as FAEDF is non-preemptive, the updated Task_Timer of the task being set-up is equal to the time left until the next scheduling point $t_{SP}$. Therefore, when the Task_Timer of a ready task is shorter than the Task_Timer of the task being set-up, then that task is to miss its set-up deadline. In this case, the task is removed from the Ready queue and its TaskID is stored in the Missed Deadlines Buffer, prior to informing the main CPU.

When a task $\theta_i$ is scheduled, at $t_{SP,i}$, its Task_Timer is loaded with an updated $t_{ICAP_i}$ value which has been computed considering the applicable performance improvements. When the data trace of $\theta_i$’s (critical) data producer task is accessed for the first time, then its voting_done flag is set to ‘1’. Additionally, the number of accesses to be performed to the results computed by the scheduled task is set equal to the amount of data consumer tasks associated to that task. If the scheduled task needs to exchange data with the main CPU, the value of the Task_Timer before update is saved for future use in the task’s data segment. Note that this value indicates the slack of the exclusively computing task, which continues to be valid for its associated exclusively communicating task.

When the Task_Timer of the task being set-up is equal to 0, then the task should be ready to start executing. If not, then HWuK is not working properly and recovery actions need to be taken (see chapter 4). Therefore, the Task_Timer of the task being set-up is
used as a watch-dog timer\textsuperscript{3}. At this point, if the task is an exclusively communicating task, the main CPU is informed that the results are ready in its IDB, and if it is an exclusively computing task, its Task_Timer is loaded with $t_{E,i} \cdot BUFR\_DIVIDE$, where BUFR\_DIVIDE accounts for the used clock rate.

When the Task_Timer of an executing task is equal to 0, then that task should have completed its computation and the main CPU is informed. Again, the tasks to be passed to the CPU are buffered in the Finished Buffer to adapt the different speeds of HWuK and main CPU. Note that to definitely confirm that the task has finished correctly, the corresponding HWS are checked and the computed results are voted when configuring the subsequent data consumer task.

A special case is when the finishing hardware task needs to exchange data with the main CPU. Then, the PicoBlaze updates the excl_comm flag in the task’s data segment, switching from the exclusively computing task to the exclusively communicating task, and restores the task’s slack value in the Task_Timer. The exclusively communicating task is appropriately re-inserted in the corresponding position within the Ready queue based on its slack, while the exclusively computing task is buffered in the Allocated queue to reflect the fact that its circuitry remains configured in the FPGA.

To ensure the correctness of the scheduler’s functioning it is mandatory that the Task_Timers of the ready tasks are updated at every $t_{KT}$. With the objective of detecting any malfunctioning as early as possible it is also advisable to update the Task_Timer of the task being set-up within this time. On the other hand, note that not updating the Task_Timers of executing tasks results in unnecessary occupation of FPGA resources, but it does not threaten the correctness of the scheduler. Updating the Task_Timers is normally completed within a small fraction of the $t_{KT}$ period, and the rest of the time is used to communicate with the main CPU, and to manage the task queues. The PicoBlaze generates an alive pulse when it finishes updating the critical Task_Timers and, the kernel timer is responsible for checking that at least one alive pulse is received from the PicoBlaze within a maximum number of kernel ticks. Otherwise, recovery actions need to be taken.

\textsuperscript{3}In newer FPGA families (e.g., Spartan-6), a watch-dog timer is included in the configuration interface.
Scheduler’s PicoBlaze evaluates the real-time tightness as it updates the Task_timers of the ready tasks. Note that: \( \sum_{v \in \text{Ready Queue}} \frac{t_{\text{ICAP}}}{t_{\text{d}} - t_{\text{ICAP}}} = \sum_{v \in \text{Ready Queue}} \frac{t_{\text{Timer}} + t_{\text{ICAP}}}{t_{\text{ICAP}}} \). To speed-up this computation the PicoBlaze is to be equipped with a hardware division module. Based on the obtained value for the real-time tightness at each time, FAEDF is enabled to track future finishing tasks or not (FA_en flag).

The FAEDF scheduling algorithm is executed when the allocator has finished updating the EAD. Indeed, prior to executing the FAEDF algorithm, the scheduler retrieves the MER(s) from the allocator. In order to improve performance, the scheduler decides alternative tasks to be scheduled in case the allocator is unable to find a suitable allocation for the previously selected task, i.e., scheduling and allocation decisions are parallelised. When a task is successfully allocated, the scheduler’s PicoBlaze updates its \((x, y)\) allocation(s) in the Task BRAM and also keeps record of which version of the task is being used; i.e., R2L, L2R, T2B or B2T. Next, the scheduler’s PicoBlaze checks which executing tasks are expected to finish by when the task is completely set-up on the device, i.e., by the next scheduling point. If any, these tasks are buffered in the Finishing Expected Buffer prior to being passed to the allocator to be considered when updating the Area Matrices. If the Ready queue is empty when executing the FAEDF algorithm or, if the latter produces a blank scheduling, the scheduler’s PicoBlaze allows the configuration manager to perform diagnosis functions (e.g., scrubbing).

The area-related functions implemented by the scheduler’s PicoBlaze are twofold. Every time the allocator decides an allocation for a task, and while the task is being set-up on the device, the scheduler’s PicoBlaze checks whether it overlaps with any task in the Allocated queue. If any, the overlapping tasks are removed from the Allocated queue and, as soon as the ICAP is available, the corresponding configuration is blanked in the configuration memory by toggling the appropriate functions in the configuration manager.

Note that the task overlapping check could also be used to monitor the allocation activities performed by the Allocator in light of increasing the diagnosis coverage of R3TOS functioning. It is indeed possible to find allocation violations when overlapping exists with tasks in the Executing queue. It is yet to be decided whether it is preferable that the overlapping check is carried out by an independent “trusted” (and simple) custom logic or by the scheduler’s PicoBlaze itself.
Chapter 6- Low-Level Hardware Support for the R3TOS HWuK

The second area-related function implemented by the scheduler’s PicoBlaze is the management of data traces. The location of a given data trace is updated in the producer task’s data segment when the data trace is moved to the IDB of a consumer task. Besides, the allocator is informed that the previous location of the data trace is available for future use. The scheduler’s PicoBlaze is also in charge of updating the number of times each data trace is accessed, or would have been accessed if the data consumer task had not missed its set-up deadline. When a data trace is not necessary anymore, i.e., it has been accessed as many times as consumer tasks are associated to it, the allocator is informed that the corresponding resources are available for future use, and the (critical) task’s voting_done flag is set to ’0’.

6.5 Allocator

This section describes how the allocation algorithms and heuristics proposed in chapter 5 are implemented using real FPGA hardware. In a general way, Snake task allocation strategy reduces the potential number of candidate allocations for (non-critical HBC) tasks and, when none of these allocations is feasible, or when dealing with critical or LBC tasks, the whole set of available allocations on FPGA needs to be evaluated using the EAC heuristic. We resort on using the EAC heuristic as we believe it is simpler to implement than EVC and still produces good quality results (see chapter 5). The allocator provides support for the aforementioned two functioning cases: the specific evaluation of some allocations which are selected based on a number of complicated factors (e.g., type of task, data trace position and connectivity offered by DRTs), and the systematic evaluation of all feasible allocations on FPGA.

As for the scheduler, aiming at a trade-off between area and speed, the allocator includes a PicoBlaze processor, which implements most of its functionality and controls some specific-purpose logic to accelerate the execution of selected operations. Namely, the latter logic includes the Architecture Checker (AC), which is in charge of dealing with resource heterogeneity when allocating a task, the Empty Area Descriptor Updater (EADU), which is responsible for updating the EAD after each task allocation, and the Allocation Quality Evaluator (AQE), which accelerates EAD consultation process when making the allocation decisions. Besides, the allocator includes a memory (FPGA State BRAM) to store the state
of the FPGA at any time, i.e., which resources are in use, which are damaged and which are available to be used.

Note that in this section we do not go into details regarding to FCCRs. It suffices to say that the FPGA State BRAM is split into independent segments to mimic the separated FCCRs on the FPGA and the same operations are repeated for each of these regions; e.g., update of the area matrices and EAD.

### 6.5.1 FPGA State BRAM

Figure 6.17 shows the internal organisation of the FPGA State BRAM, where both the EAD and the 2DAM matrix are held. The BRAM is configured to be 8-bits width.

The FPGA state is kept within the 0x000 to 0x03B segment, using 2 bits to represent the state of each FPGA location: the most significant bit indicates whether the resources included in that location are damaged (‘1’) or not, and the least significant bit indicates if that location is available to be used (‘0’) or already assigned to a task (‘1’). Thus, each memory position describes the state of up to 4 FPGA locations.

A separate space of 256 memory positions is reserved for each area matrix defined in chapter 5: the LAM matrix is held in positions 0x040 to 0x139, the RAM matrix in positions 0x140 to 0x239, the ULAM matrix in positions 0x240 to 0x339, the DLAM matrix in positions 0x340 to 0x439, the URAM matrix in positions 0x440 to 0x539, the DRAM matrix in positions 0x540 to 0x639, and the 2DAM matrix is kept in positions 0x640 to 0x739. An 8-bit memory word is dedicated to each FPGA location in the area matrices, allowing to deal with a maximum of 256 FPGA locations without overflow. These locations can be arbitrarily used to cover a wide range of different FPGA devices, ranging from $H_x=64 \times H_y=4$ to $H_x=21 \times H_y=12$. Overflow can occur when computing the 2DAM matrix, in which case the highest value (255) is written. Being part of the EAD, up to 64 memory positions are reserved to store the column_MERs, in positions 0x740 to 0x779.

Considering that the largest Virtex-4 FPGA has 12 clock regions, and there are two BUFRs and two regional clock lines per clock region, 24 memory positions are reserved for storing the clocking configuration (i.e., in positions 0x780 to 0x798). The information stored for each BUFR includes the BUFR_DIVIDE parameter, the clock input source (i.e., RCMUX configuration), and a bit to indicate whether the BUFR is damaged (‘1’) or not (‘0’).
Figure 6.17: Organisation of the FPGA State BRAM
Finally, memory positions 0x799 to 0x7AF are used to receive the list of the damaged resources, which are identified by the configuration manager when carrying out the diagnostic test presented in chapter 4. The remaining positions in the FPGA State BRAM (i.e., 0x7B0 to 0x7FF), is used to implement the stack of the allocator's PicoBlaze.

### 6.5.2 Empty Area Descriptor Updater (EADU)

As pointed out in chapter 5, the amount of time available to update the EAD without degrading the performance is limited by the duration of the set-up phase of the tasks, i.e., EAD is to be updated in parallel with task setting-up through the ICAP. In order to speed-up the EAD updating, which is indeed a time-consuming process whose complexity increases with FPGA's size, the HWuK includes a specific logic (EAD Updater) that is coupled to the two ports of the FPGA State BRAM. This logic is very easy to control with only two interface signals: input `start` updating and output `end` updating. These two signals are driven by allocator's PicoBlaze, which also controls access to the FPGA State BRAM. Indeed, note that the latter BRAM is shared between the PicoBlaze, EADU, AQE and configuration manager, which provides the list of detected damaged resources in the chip upon request by the allocator's PicoBlaze. Prior to toggling the EADU, the `FPGA_state` is renewed by the allocator's PicoBlaze considering both the damaged resources and the executing tasks which are expected to finish by the next scheduling point.

The performance improvement brought about by the EADU is based on both parallelism and pipelining. Indeed, the EADU proceeds in four phases as shown in Figure 6.18. In a first phase, the `RAM` and `LAM` matrices are computed in parallel using the information included in the `FPGA_state` memory segment, which is simultaneously accessed through the double port of the BRAM. As the only difference when computing the `RAM` and `LAM` matrices is that `FPGA_state` is scanned in opposite directions (i.e., right to left and left to right), the amount of time needed to compute both matrices is the same. Afterwards, the four area matrices `ULAM`, `DLAM`, `URAM` and `DRAM` are computed in pairs using the two ports of the BRAM and the information included in the `RAM` and `LAM` memory segments. Namely, while the `ULAM` matrix is computed through the port A, the `URAM` matrix is computed through the port B, and then, `DLAM` and `DRAM` matrices are simultaneously computed through each BRAM port. Note that when computing the `DRAM` matrix, the `column_MERs` are also up-
dated. In the fourth and last phase, the 2DAM matrix is updated using the area matrices computed during the previous phase. Notably, this computation is speeded-up by 2, as the values of two area matrices can be simultaneously accessed through the two ports of the FPGA State BRAM.

\[ \text{Figure 6.18: EAD updating process} \]

The EADU consumes 284 slices, 451 LUTs and 226 flip-flops in the FPGA, and in the situation described in chapter 5 (i.e., \( H_x = 15 \) and \( H_y = 12 \)), it allows for up to 10x speed improvement with regard to the solely PicoBlaze-based software implementation reported in [Hong et al., 2011].

### 6.5.3 Architecture Checker (AC)

Figure 6.19 shows the structure of the AC, whose main objective is to rapidly check whether it is feasible to allocate a task on a given FPGA location in terms of types of resources, saving much computational cost to the allocator’s PicoBlaze. The latter controls the AC block by means of two signals: \textit{shift} and \textit{match}. Note that this block is mainly used to find feasible allocations when systematically evaluating all of the candidate positions for a task. On the other hand, when using \textit{Snake}, the compatibility of FPGA’s layout and task’s internal architecture is checked by the PicoBlaze as the number of allocations to evaluate is reduced.

The central part of the AC module is a shift register (\texttt{ad}) of depth equal to \( H_x \) (i.e., amount of columns in the FPGA). Another register (\texttt{AD}) is used to store the architecture
descriptor of the FPGA device itself. Each cell in these registers accounts thus for a resource column, with the type of resource coded using 2 bits: “00” for CLBs, “01” for DSP48s, “10” for BRAMs, and “11” for other resources (e.g., PowerPC and IOBs). The architecture descriptor of the task to check $\theta_i$ is loaded in the $ad$ register, and shifted cell by cell to cover all of the possible allocations for it, i.e., until the task descriptor reaches the deepest $H_x$ cell in the shift register. To check whether the type of resources required by the task matches with the FPGA resources actually available in each position, both $ad$ and $AD$ cells are XORed, where '0' means that the type of all of the resources is the same. The cells are individually enabled to take part in the XOR operation in order to exclude the resources which are not actually used by the task in each checked position. To do this a shift register ($EN$) with the same depth of $ad$ and $AD$ is used. This register is initially loaded with all zeros, except for the cells occupied by the task descriptor, which are loaded with '1'; i.e., $EN(i)=1'$ $\forall$ $i \leq h_{x,i}$ and $EN(i)=0'$ $\forall$ $i > h_{x,i}$. The sequence of '1's is shifted in the $EN$ register when the task descriptor is shifted in the $ad$ register to reflect which columns are to be used by the task in the checked position at any time. Therefore, the PicoBlaze is only responsible for controlling the shift in the registers and checking the feasibility of the placement. The latter is given by the match signal, where $match='1'$ means that allocation is feasible.

Figure 6.20 shows the resource requirements of the AC module for different values of $H_x$. For comparative purposes, note that the AC requires a similar amount of slices as required by a PicoBlaze when the FPGA device has 45 columns. The upper bound speed improvement brought about by this module when checking the allocatability of a task $\theta_i$ compared to a solely PicoBlaze-based software implementation can be roughly estimated to be around $2 \cdot h_{x,i}$. Indeed, without using the AC, 6 PicoBlaze instructions are needed to check the resource compatibility in each FPGA column: 2 instructions for accessing the FPGA descriptor, another 2 instructions for accessing the task descriptor, 1 instruction to compare both values and another instruction to update the next column to be checked. On the other hand, when using the AC, only 3 PicoBlaze instructions are required to perform this check: 2 instructions to enable and disable the shift signal, and another instruction to input the match value. Assuming the worst-case where all resource columns are compatible, the achieved 2x speed-up factor brought about by the AC when checking the resource compatibility of one column is extended to the $h_{x,i}$ columns the task spans.
Figure 6.19: Simplified structure of the Architecture Checker (AC)

Figure 6.20: Resource requirement of the Architecture Checker for different FPGA sizes $H_x$
6.5.4 Allocation Quality Evaluator (AQE)

When toggled by the allocator's PicoBlaze, the AQE takes over access to the FPGA State BRAM to quickly compute the quality of candidate allocations. This is done by adding the EAC scores stored in the 2DAM segment that correspond with the FPGA positions to be assigned to the task in each evaluated allocation. The latter computation is automatically performed by the AQE based on the allocation and task information passed by the PicoBlaze (i.e., $h_{x,i}$, $h_{y,i}$, $X_{allo}$ and $Y_{allo}$). Besides, the AQE keeps track of the quality of the checked allocations, indicating to the PicoBlaze which is the one that produces the least fragmentation on the FPGA by means of the best signal (i.e., lowest sum of EAC scores). It is important to note that the AQE does not check the feasibility of the allocations. This is done by the allocator's PicoBlaze by consulting the EAD and using the AC.

The interface of the AQE consists of 4 control signals ($rst$, $start$, $end$, $best$) as well as an 8-bit input to receive the allocation and task parameters.

The AQE consumes 33 slices, 53 LUTs and 48 flip-flops in the FPGA and allows for a significant speed improvement when making the allocation decisions with regard to the solely PicoBlaze-based software implementation reported in [Hong et al., 2011]. As shown in Figure 6.21, the achievable acceleration increases with the size of the task to allocate until it reaches a high bound of about 9x. This behaviour is due to two reasons. First, when dealing with big tasks, the communication overhead between PicoBlaze and AQE is smaller because there are fewer candidate allocations to check. Second, evaluating the allocation quality of a big task requires more computations (i.e., additions) to be done, which are indeed accelerated by the AQE.

![Figure 6.21: Acceleration factor achieved by the AQE for different task sizes](image-url)
6.5.5 The Allocator’s PicoBlaze

The allocator’s PicoBlaze (1) updates the information kept in the FPGA_state and clocking configuration segments of the FPGA State BRAM, (2) drives the EADU, (3) searches candidate allocations using the results provided by the AC and the information stored in the EAD, (3) implements the Snake task allocation strategy and, (4) makes the final allocation decisions using the results provided by the AQE.

The allocator’s PicoBlaze receives the information related to a task \( \theta_i \) to be allocated from the scheduler’s PicoBlaze. The transmitted parameters include \( h_{x,i}, h_{y,i}, \) task’s architecture descriptor, highest usable clock rate, and \((x, y)\), in case \( \theta_i \) still remains allocated on the FPGA. Additionally, depending whether Snake is applicable (i.e., the task is HBC and not critical) other parameters are also transmitted, including \( t_{A,i}, t_{D,i}, t_{R,j} \) (where \( \theta_j \) is the data producer task of \( \theta_i \)), and location of the data trace to be processed. For instance, there is no need to transmit the location of the data trace when allocating LBC tasks. The information received from the scheduler is used to find the most convenient allocation(s) for \( \theta_i \) within the FPGA. Specifically, LBC tasks are allocated on the position where they can be fed with the highest clock rate (in order to speed-up task’s execution phase), critical HBC tasks are allocated on the position where they minimise fragmentation on the device, and non-critical HBC tasks are allocated so that data traces can be accessed directly from the memories where they are stored or else by using a DRT (in order to speed-up task’s set-up phase).

As soon as finding an allocation for a task, the PicoBlaze updates the information stored in the clocking configuration segment of the FPGA State BRAM. Then, the FPGA_state segment is amended using the information delivered by both scheduler and configuration manager. The former provides the list of the tasks which are expected to finish by the next scheduling point as well as list of tasks that have been deallocated as they overlap with the task being allocated. The latter provides the list of the detected damaged resources in the chip. Immediately after updating the FPGA_state, the PicoBlaze enables the EADU to renew the EAD and area matrices in the FPGA State BRAM.

A simplified functioning diagram of the allocator’s PicoBlaze is outlined in Figure 6.22.
6.6 Configuration Manager

The configuration manager is responsible for interacting with the FPGA’s configuration interface, transmitting the appropriate configuration commands to it to perform the operations dictated by the scheduler and allocator.

As with the scheduler and allocator, we use a PicoBlaze to coordinate the different circuitry that makes up the configuration manager. This circuitry includes: (1) an ICAP driver, which is able to exploit the full reconfiguration bandwidth offered by the ICAP (i.e., it transmits or receives a 32-bit data word per clock cycle), (2) a CRC32 module, which is able to compute a 32-bit CRC over the data read from the ICAP without constraining the perfor-
mance, (3) the STARTUP support logic, and (4) a Bitstream BRAM, to temporarily store the read-back data as well as the configuration commands prior to transmission to the ICAP. Additionally, the configuration manager includes a Configuration Guardian (CG) to trap erroneous accesses to the ICAP, and a direct link to the external memory where the tasks’ partial bitstreams are stored.

6.6.1 Bitstream BRAM

Figure 6.23 shows the internal organisation of the Bitstream BRAM, which is configured as a 512 x 32-bit dual-ported memory to match the ICAP’s data bus width. One of the BRAM ports is used to load the partial bitstreams of the tasks, which is bulked from the external memory by the memory controller, and the other port is accessed either by the PicoBlaze or by the ICAP driver, when transferring data to/from the ICAP.

The Bitstream BRAM holds a set of pre-defined sequences of configuration commands at memory positions 0x004 to 0x0A8. Examples of these sequences are the Configuration Operation Starting Sequence (COSS), which sets-up the ICAP, or the Configuration Operation Ending Sequence (COES), which desynchronises the ICAP. These sequences are the substrate to build a set of basic functions (e.g., read/write a single or a subset of frames from / to the FPGA configuration memory or blank a specific region on the FPGA), upon which R3TOS HWuK services are implemented. Indeed, the command sequences are conveniently used by the PicoBlaze as raw “configuration templates” which are adjusted with the specific parameters of the operation that needs to be performed at any time prior to sending them to the ICAP. Writing to the configuration templates is not permitted except in the positions that need to be adjusted. Furthermore, since the PicoBlaze needs to know some information about the FPGA architecture when using the command templates (e.g., how many minor frames are in each FPGA column when blanking a specific region), an FPGA architecture descriptor is stored in the 4 lowest memory positions of the Bitstream BRAM.

The rest of the Bitstream BRAM positions, except those which implement the stack of the PicoBlaze, are used to temporarily store the configuration information to be loaded to the FPGA configuration memory. Buffering configuration information is needed as most of the ICAP operations consist in writing a piece of configuration data which has been previously read-back from the FPGA memory itself; e.g., enable a HWS, scrub a soft-error, or
Figure 6.23: Organisation of the Bitstream BRAM
relocate data from a producer task’s ODB to a consumer task’s IDB. Indeed, the buffers to
temporarily store read-back configuration information are strategically located to ease the
subsequent writing operations: (1) there is space to embed the content of one read-back
frame in the middle of the template used for writing a single frame to the configuration
memory (at memory positions 0x068 to 0x091) and, (2) the commands to write multiple
configuration frames are located immediately before the high capacity buffer where up to
8 frames can be buffered (located at memory positions 0x0A9 to 0x1F1). The only opera-
tion where the data written to the FPGA has not been previously read-back is when a task
is configured for the first time. In this case, the configuration information is bulked to the
bitstream BRAM, namely to the high capacity buffer, from the external memory through one
of the memory ports and the ICAP driver accesses this information some clock cycles after
through the other port. The high capacity buffer is thus used to temporarily store either
partial bitstreams or a set of read-back frames.

**Configuration Operation Starting Sequence (COSS)**

Figure 6.24 shows the configuration commands to be transmitted to the ICAP prior to start-
ing any configuration operation. These commands synchronise the ICAP and disable the
CRC checking. Note that CRC checking must be disabled as the HWuK operates in the con-
figuration layer of the FPGA and hence, it intentionally modifies the configuration infor-
mation. The modifiable configuration information stored in the Bitstream BRAM is to be
protected by the ECC codes of the BRAM itself.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA995566</td>
<td>Sync. Word</td>
</tr>
<tr>
<td>20000000</td>
<td>NO-OP</td>
</tr>
<tr>
<td>30008001</td>
<td>Write 1 Word to CMD Register</td>
</tr>
<tr>
<td>00000007</td>
<td>RCRC Command</td>
</tr>
<tr>
<td>20000000</td>
<td>NO-OP</td>
</tr>
<tr>
<td>20000000</td>
<td>NO-OP</td>
</tr>
<tr>
<td>30012001</td>
<td>Write 1 Word to COR Register</td>
</tr>
<tr>
<td>10042FDD</td>
<td>Bypass CRC check</td>
</tr>
</tbody>
</table>

*Jump to corresponding configuration template (Figure 6.25 / 6.26 / 6.27 / 6.28)*

**Figure 6.24**: COSS: Configuration Operation Starting Sequence
Frame Reading-back Configuration Template

Figure 6.25 shows the configuration template for reading-back a subset of frames from the FPGA configuration memory. This template includes three parameters to be adjusted by the PicoBlaze: (1) the address of the first frame to be read-back (FAR), (2) the amount of words to be read-back, including the pad words, and (3) whether it is a diagnosis oriented read-back or not. Indeed, when the read-back operation is used to detect soft-errors, the user information stored in LUTs (e.g., LUT-RAMs or SRL16s) is masked out with zeros by setting GLUTMASK_B bit to ‘0’ in the Control Register (CTL). Otherwise, the actual user information is read-back by setting the GLUTMASK_B bit to ‘1’ (e.g., when polling a HWS).

| 30008001 | Write 1 Word to CMD Register |
| 00000004 | RCFG Command |
| 3000C001 | Write 1 Word to MASK Register |
| 0000100 | Permit writing to GLUTMASK_B bit |
| 3000A001 | Write 1 Word to CTL Register |
| 0000x00 | GLUTMASK_B=x |
| 30002001 | Write 1 Word to FAR Register |
| xxxxxxxx | FAR=xxxxxxxx |
| 28006xxx | Read xxx Words from FDRO Register (Type 1) |
| 20000000 | NO-OP |

*Download configuration data to a temporal configuration information buffer*
*Jump to COES (Figure 6.29)*

Figure 6.25: Configuration template for reading-back a subset of frames from the FPGA's configuration memory (Note: the value of "x" is adjusted by the PicoBlaze)

Frame Writing Configuration Templates

Figure 6.26 and Figure 6.27 show the configuration templates for writing a single and multiple frames to the FPGA configuration memory, respectively. In both cases, the device ID-CODE is written in the appropriate register prior to transmitting actual configuration data to the ICAP. This IDCODE is initialised in the Bitstream BRAM at design time based on the target FPGA device. Furthermore, as for the read-back configuration template, the GLUTMASK_B bit must be handled appropriately, i.e., it must be set to ‘0’ when scrubbing a soft-error and to ‘1’ in the rest of the cases.
Figure 6.26: Configuration template for writing a single frame to the FPGA's configuration memory or for writing the same input data to the three redundant instances of a quasi-FTU task (Note: the value of “x” is adjusted by the PicoBlaze, the value of “c” is set at design time based on the target FPGA device and, “d” refers to the data that has been previously read-back from the configuration memory)
<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30018001</td>
<td>Write 1 Word to IDCODE Register</td>
<td>IDCODE = ccccccccc</td>
</tr>
<tr>
<td>3000C001</td>
<td>Write 1 Word to MASK Register</td>
<td>Permit writing to GLUTMASK_B bit</td>
</tr>
<tr>
<td>3000A001</td>
<td>Write 1 Word to CTL Register</td>
<td>GLUTMASK_B = x</td>
</tr>
<tr>
<td>30002001</td>
<td>Write 1 Word to FAR Register</td>
<td>FAR = xxxxxxxx</td>
</tr>
<tr>
<td>30004xxx</td>
<td>Write xxx Words to FDRI Register (Packet 1)</td>
<td>Upload configuration data from the high capacity buffer (addr. 0x0A9)</td>
</tr>
</tbody>
</table>

*Figure 6.27:* Configuration template for writing a subset of frames to the FPGA’s configuration memory (Note: the value of “x” is adjusted by the PicoBlaze and the value of “c” is set at design time based on the target FPGA device)

As shown in Figure 6.26, MFWR commands are used for writing single frames (e.g., enable a HWS or scrub a soft-error). This method is faster than using the traditional way (Figure 6.27), i.e., writing to the Frame Data Register Input (FDRI), as there is no need to add any extra pad information to the frame data [Claus et al., 2006]. In total, up to 37 clock cycles are saved per written frame. The frame data is loaded in the single frame buffer within the Bitstream BRAM. However, MFWR commands can only be used in a frame-by-frame fashion and therefore, it is preferable to use the traditional way when writing a large set of frames to a single position on the FPGA (e.g., when copying the 64 BRAM content frames to a single instance of a task). Since there is space to buffer up to 8 frames in the Bitstream BRAM, only 8 pad frames need to be read-back when accessing the data producer task’s ODB; i.e., 64/8. Besides, MFWR commands are also preferred when writing the same set of frames to multiple positions on the FPGA (e.g., when copying the BRAM content frames to the three redundant task instances of a quasi-FTU). In this case, the pad frame overhead is compensated with the capability to write the same frame to three locations at the same time.

When uploading configuration data to the FPGA configuration memory using the FDRI register, the PicoBlaze needs to adjust the address of the first frame to be written (FAR) as well as the number of words to be written, including the pad words. On the other hand, when using MFWR commands, the PicoBlaze needs to adjust the address of the three target frames to be written (FAR1, FAR2 and FAR3). In case there is only one frame to be written, the PicoBlaze only needs to define FAR1.
Frame Blanking Configuration Template

The configuration template for blanking a subset of frames in the FPGA configuration memory is shown in Figure 6.28. As an all-zeros frame is to be written to several frames, MFWR commands are used. The PicoBlaze adjusts the address of the first frame to be blanked (FAR) and the amount of frames to be blanked. Typically, blanking applies on an entire FPGA column: the first frame to be blanked has a MINOR=0 and the amount of frames to be blanked is equal to the amount of frames in that FPGA column. Hence, the ICAP driver automatically increases the MINOR field without modifying the rest of the fields of the frame address (see the loop in Figure 6.28).

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30018001</td>
<td>Write 1 Word to IDCODE Register</td>
</tr>
<tr>
<td>cccccccc</td>
<td>IDCODE=ccccccc</td>
</tr>
<tr>
<td>30008001</td>
<td>Write 1 Word to CMD Register</td>
</tr>
<tr>
<td>00000001</td>
<td>WCFG Command</td>
</tr>
<tr>
<td>20000000</td>
<td>NO-OP</td>
</tr>
<tr>
<td>30002001</td>
<td>Write 1 Word to FAR Register</td>
</tr>
<tr>
<td>xxxxxxxxx</td>
<td>FAR=xxxxxxx</td>
</tr>
<tr>
<td>20000000</td>
<td>NO-OP</td>
</tr>
<tr>
<td>30004029</td>
<td>Write 41 Words to FDRI</td>
</tr>
<tr>
<td>00000000</td>
<td>1st blank word</td>
</tr>
<tr>
<td></td>
<td>... 39 blank words more</td>
</tr>
<tr>
<td>00000000</td>
<td>41st blank word</td>
</tr>
<tr>
<td>30002001</td>
<td>Write 1 Word to FAR Register</td>
</tr>
<tr>
<td>xxxxxxxxx</td>
<td>FAR=xxxxxxx</td>
</tr>
<tr>
<td>30008001</td>
<td>Write 1 Word to CMD Register</td>
</tr>
<tr>
<td>00000002</td>
<td>Write MFWR Command</td>
</tr>
<tr>
<td>20000000</td>
<td>NO-OP</td>
</tr>
<tr>
<td>loop:</td>
<td></td>
</tr>
<tr>
<td>30014002</td>
<td>Write 2 Words to MFWR Register</td>
</tr>
<tr>
<td>00000000</td>
<td>Write 1st Dummy Word</td>
</tr>
<tr>
<td>00000000</td>
<td>Write 2nd Dummy Word</td>
</tr>
<tr>
<td>30002001</td>
<td>Write 1 Word to FAR Register</td>
</tr>
<tr>
<td>rrrrrrrr</td>
<td>FAR=rrrrrrr</td>
</tr>
<tr>
<td>end loop:</td>
<td></td>
</tr>
</tbody>
</table>

Jump to COES (Figure 6.29)

Figure 6.28: Configuration template for blanking a subset of frames in the FPGA configuration memory (Note: the value of “x” is adjusted by the PicoBlaze, the value of “r” is updated by the ICAP driver and, the value of “c” is set at design time based on the target FPGA device)
Configuration Operation Ending Sequence (COES)

Figure 6.29 shows the configuration commands to be transmitted to the ICAP after every configuration operation. As the CRC checking is disabled in the COSS sequence, the default CRC value is transmitted to the ICAP (i.e., 0x0000DEFC in the case of Virtex-4). Furthermore, the GLUTMASK_B bit in the CTL register is always reset to the default value (i.e., equal to ‘1’), and finally, the ICAP is desynchronised.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000C001</td>
<td>Write 1 Word to MASK Register</td>
</tr>
<tr>
<td>0000100</td>
<td>Permit writing to GLUTMASK_B bit</td>
</tr>
<tr>
<td>3000A001</td>
<td>Write 1 Word to CTL Register</td>
</tr>
<tr>
<td>0000100</td>
<td>GLUTMASK_B=1</td>
</tr>
<tr>
<td>30000001</td>
<td>Write 1 Word to CRC Register</td>
</tr>
<tr>
<td>0000DEFC</td>
<td>CRC Predetermined value when CRC is bypassed</td>
</tr>
<tr>
<td>30008001</td>
<td>Write 1 Word to CMD Register</td>
</tr>
<tr>
<td>0000000D</td>
<td>DESYNCC Command</td>
</tr>
<tr>
<td>20000000</td>
<td>NO-OP</td>
</tr>
<tr>
<td>20000000</td>
<td>NO-OP</td>
</tr>
</tbody>
</table>

Figure 6.29: COES: Configuration Operation Ending Sequence

6.6.2 ICAP Driver

Besides commanding the configuration data transfers between the Bitstream BRAM and the ICAP, the ICAP driver is responsible for dealing with all of the particularities of FPGA hardware described in chapter 2. The data transfers are carried out in 32-bit words and at 100 MHz clock rate. This results in an effective reconfiguration bandwidth as high as 380 MB/s when working with small partial bitstreams (e.g., less than 10 KBs) and reaches 390 MB/s when working with large partial bitstreams (e.g., tens/hundreds of KBs).

As shown in Table 6.3, the ICAP driver implements up to 6 basic functions, from which 5 are directly supported by the configuration templates described in section 6.6.1: (1) Read-Back Frame(s) (RBF) function refers to the template shown in Figure 6.25, (2) Write a Single Frame to a Single location (WSF2S) function is based on the template depicted in Figure 6.26, (3) Write a Single Frame to Multiple locations (WSF2M) function uses the template shown in Figure 6.26, (4) Write Multiple Frames to a Single location (WMF2S) function refers to the template shown in Figure 6.27 and, (5) Blank Frame(s) (BlF) function executes the com-
mand sequence depicted in Figure 6.28. The sixth function, *Partial Bitstream (Re)allocation* (PBR), does not rely on any pre-defined configuration template, it consists in transmitting the configuration data stored in the high capacity buffer to the ICAP. The ICAP driver keeps a busy signal high while executing any of these functions to indicate to the PicoBlaze that the Bitstream BRAM is not accessible.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Template</th>
<th>Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBF</td>
<td>Read-back a subset of frames</td>
<td>Figure 6.25</td>
<td>FAR, # words</td>
</tr>
<tr>
<td>WSF2S</td>
<td>Write a single frame to a single location</td>
<td>Figure 6.26</td>
<td>FAR</td>
</tr>
<tr>
<td>WSF2M</td>
<td>Write a single frame to multiple locations</td>
<td>Figure 6.26</td>
<td>FAR1, FAR2, FAR3</td>
</tr>
<tr>
<td>WMF2S</td>
<td>Write multiple frames to a single location</td>
<td>Figure 6.27</td>
<td>FAR, # words</td>
</tr>
<tr>
<td>BIF</td>
<td>Blank a subset of consecutive frames</td>
<td>Figure 6.28</td>
<td>FAR, # frames</td>
</tr>
<tr>
<td>PBR</td>
<td>Relocate a partial bitstream</td>
<td>-</td>
<td>(X,Y), Reg. Clk Line</td>
</tr>
</tbody>
</table>

*Table 6.3:* Basic functions implemented by the ICAP driver

The ICAP driver is structured around a central FSM which directs the functioning and generates the signals to control the ICAP and the Bitstream BRAM. Specifically, the FSM transfers the configuration command flow to the Bitstream BRAM positions where the configuration commands required at any time are stored and, in the case of RBF functions, it also commands the writing of the read-back data in the latter memory. When a single frame is read-back, the data delivered by the ICAP is saved in the single-frame buffer and when multiple frames are read-back, the data is saved in the high capacity buffer. Note that the ICAP PicoBlaze can access the read-back data buffered in the Bitstream BRAM.

When executing the PBR function, the ICAP driver acts as a bitstream filter. It dynamically manipulates the frame addresses and regional clock line connections when transferring the task's partial bitstream stored in the high capacity buffer to the ICAP. This buffer is cyclically accessed, i.e., after the last position in the buffer is read (0x1F1), the next data to be transmitted is accessed in the first position (0x0A9).

The ICAP driver is responsible for translating the (X,Y) allocation scheme used by the scheduler and allocator into the suitable FPGA addressing scheme. Thus, the relocated frame addresses are computed taking the original frame addresses in the partial bitstream and the target (X,Y) position as inputs. It is important to note that the frame addresses of the partial bitstreams are manipulated to be referred to the FPGA's upper-left corner prior
to being loaded in the external bitstream memory; i.e., the frame addresses are relative to
the partial bitstreams. The row field is computed based on Y coordinate, while the column
field is computed based on X coordinate. Due to the different column addressing scheme
for BRAMs and CLBs/DSP48s, two indexes are derived from X: $X_{BRAM}$ and $X_{CLB/DSP}$. In or-
der to give support for MFWR-based Triple Modular Redundancy (TMR), the ICAP driver
allows up to three different target locations for (critical) tasks. As it cannot manage more
target locations in the current implementation, its use in the scope of high-performance
applications to allocate multiple cloned tasks is limited.

In order to account for the frame mirroring effect between top and bottom halves of the
FPGA, the ICAP driver automatically reverses the content of any frame when it is written or
read to/from the bottom half of the device. This is done by reading or writing the frame
words from/to the Bitstream BRAM in reverse order, i.e., from the 1st to the 41st (when
working on the top half) and from the 41st to the 1st (when working on the bottom half),
and also reversing the bits within each word, i.e., conceptually: $\text{output}(0 \text{ to } 31) \leq= \text{input}(31 \text{ downto } 0)$, except for the central 21st HCLK word. Note that when a frame is
read from the bottom and it is again written to the bottom, both reversing operations cancel
each other. On the other hand, when the source and target frame locations are opposite, the
reversing operation applies only once.

While transmitting the configuration information to the ICAP, the FSM processes the
packet headers to distinguish between configuration commands and frame data. This is
necessary to control the amount of words that are read or written and, to append and dis-
card the pad information when required. As shown in Figure 6.30, when performing an RBF
function, the driver only copies the actual configuration data to the buffers, and when car-
rying out a write function (WSF2S, WSF2M and WMF2S), the driver automatically transmits
the required extra pad information to the ICAP.

Besides processing the packet headers, the FSM detects two specific sequences of con-
figuration commands. The logic to manipulate the frame address is activated in PBR func-
tions immediately after detecting a write to the FAR register (i.e., 0x30002001), and the con-
figuration data transfer is finished when a DESYNC command is detected (i.e., 0x30008001
followed with 0x0000000D). Note that any intermediate NO-OPs in-between are discarded
in both cases (i.e., 0x00000000 or 0x20000000). Additionally, the FSM keeps track of the
frame address and of the frame word being transferred at any time to: (1) disable the SAVE-
DATA bits when writing BRAM content frames, (2) enable data reversing when working on
frames located on the bottom half of the device, and (3) select the required regional clock
line when allocating a task. As explained in chapter 2, SAVEDATA bits are located in the
5th, 15th, 26th and 36th words in top half frames and in the 6th, 16th, 27th and 37th words
in bottom half frames, while the regional clock selection bits are located in the 21st frame
word.

Finally, in order to overcome the coupling problem in the Frame_ECC logic of Virtex-4
FPGAs (i.e., the Frame_ECC logic works one clock cycle ahead of the ICAP), the ICAP driver
captures the ECC syndrome when reading the 40th frame word, instead of waiting to the
SYNDROMEVALID pulse which is given by the Frame_ECC logic and thus comes one clock
cycle after. Besides, the ICAP and Frame_ECC logic must be synchronised prior to check-
ning the ECC code of any frame. As shown in Figure 6.31, $41 - N_{Rd}$ words must be read to
re-synchronise both parts, where $N_{Rd}$ is the amount of read operations performed since the
last synchronisation and is reset to 0 every time it reaches 41. We note that this solution per-
mits to avoid the use of the Xilinx SEU controller macro, resulting in a simpler configuration
manager implementation.
The ICAP driver consumes 507 slices, 728 LUTs and 465 flip-flops in the FPGA. However, it is important to note that currently the driver is not optimised and thus, important resource savings are expected in the final R3TOS implementation.

### 6.6.3 STARTUP Support Logic

As introduced in chapter 4, the STARTUP primitive must be carefully used not to corrupt any part of the system while accessing the GSR signal, which is internal to FPGA's configuration logic. Namely, we have noticed that BRAMs give all zeros while the GSR signal is active. A possible explanation for this might reside in the fact that BRAMs are coupled with registers at their output. As a result, the STARTUP primitive cannot be directly managed by a PicoBlaze, which relies on BRAMs to execute its program, but by a specific logic. This logic basically generates a 2 clock cycles GSR duration pulse upon request (ini_FF signal) from the configuration manager’s PicoBlaze. Moreover, the latter PicoBlaze is also responsible
for appropriately configuring the flip-flop masking bits in order to restrict the initialisation operation to the flip-flops contained within a specific FPGA region.

It must be noted that all zeros read from BRAMs while enabling the GSR signal are decoded as LOAD s0, 0 instruction by the PicoBlaze and hence, the value stored in the s0 register must be saved and restored immediately before and after triggering the GSR signal. As shown in Figure 6.32, the code to perform these operations gives rise to a specific Interruption Service Routine (GSR_ISR) in the R3TOS PicoBlazes. Therefore, the ini_FF signal is used as interruption in all of the PicoBlazes in the HWuK, and the delay introduced by the STARTUP support logic when generating the GSR pulse allows enough time for the PicoBlazes to enter the GSR_ISR.

--- Conf. Man. PicoBlaze --
; Disable CPU's clock
; Enable HW tasks' clock
; Enable Initialise FFs
LOAD s0, s0
LOAD s0, s0
; Disable Initialise FFs
; Enable CPU's clock
GSR_ISR:
; Store s0, s0 stack
LOAD s0, s00
LOAD s0, s0
FETCH s0, s0, s0 stack
RETURN
; Enable
ADDRESS 3FF
; Check Initialise FFs
JUMP GSR_ISR

Figure 6.32: STARTUP support logic
Besides, the zeros read from the BRAMs while enabling the GSR signal can lead to undesired instruction execution in the main CPU (e.g., MicroBlaze), which might be harmful for the system. In order to secure the main CPU, the configuration manager’s PicoBlaze keeps its clock stopped while performing the GSR operation. Likewise, the clock delivered to the hardware tasks which use BRAMs is also stopped prior to activating the GSR signal. Both the clock delivered to the main CPU and to the tasks are again resumed by configuration manager’s PicoBlaze when returning from GSR_ISR.

Finally, we note that the use of STARTUP primitive could be circumvented if the bitstreams of all tasks were generated using the RESET_AFTER_RECONFIG constraint. However, this is not always up to the developer (e.g., when using third-party IP cores).

### 6.6.4 Configuration Guardian (CG)

The objective of the CG is to prevent the corruption of the system due to erroneous accesses to the ICAP. Besides, the CG increases the error detection coverage as the erroneous ICAP accesses are typically the result of a wrong configuration state of R3TOS itself.

The CG circuit stores in four (only readable) registers the coordinates of the upper-left and bottom-right vertices of the region where the R3TOS circuitry is implemented on the FPGA, \((x_{RUL}, y_{RUL})\) and \((x_{RDR}, y_{RDR})\), respectively. The latter registers are initialised with the appropriate values at design time. Since access to any frame within this region is only allowed in privileged mode, any attempt to access this region without having activated the privileged mode first is considered erroneous. Likewise, when in privileged mode, any access to a frame out of the region assigned to R3TOS is considered erroneous as well. Besides, the CG circuit includes four writable registers to load the coordinates of the upper-left and bottom-right vertices of the FPGA region assigned to the task to be accessed, \((x_{TUL}, y_{TUL})\) and \((x_{TDR}, y_{TDR})\), respectively. These coordinates can be easily computed as: \(x_{TUL} = x\), \(y_{TUL} = y\), \(x_{TDR} = x + h_x\), and \(y_{TDR} = y + h_y\), where \((x, y)\) are the allocation coordinates saved in the data segment of the task (within the Task BRAM) if it is already allocated on the device, or the values given by the allocator if the task is being allocated for the first time. The CG snoops the data transmitted through the ICAP data bus to detect any writing operation to the FAR (i.e., 0x30002001 value), and to capture the subsequent frame addresses, which are translated to \((X, Y)\) scheme in order to be compared with the coordinates stored in the
aforementioned registers. If the subsequent ICAP access is to a frame out of the specified region, then that access is considered erroneous. In this case, an output flip-flop is permanently set to '0' to trigger a full FPGA reconfiguration via its PROG_B pin.

In the unlikely case that the 0x30002001 value transmitted to the ICAP is a raw configuration data and does not correspond to a FAR writing operation, the CG may produce a false-positive error detection (with probability $\frac{1}{2^{32}}$). To avoid false-positive error detections, the CG should be able to process the header of the configuration data packets, which would make it more complex and thus more prone to error, i.e., to produce false-negatives. We think it is preferable to use a simpler CG (it currently fits in only 38 slices).

### 6.6.5 CRC32 Module

The objective of the CRC32 module is to represent every set of results computed by a task with a more manageable 32-bit CRC value, which simplifies the voting process. The CRC32 module has a parallel architecture which is based on the standard generator polynomial:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

4. The internal registers of the CRC32 module are clocked with the same clock connected to the ICAP and they are only enabled (by the ICAP driver) while the ICAP delivers valid configuration data at its DOUT port. That is, the CRC32 module is not operating while the configuration commands are transmitted to the ICAP nor while the pad information is received from it. Note that as the redundant instances of the tasks are identical, there is no need to mask out the configuration information included in the frames where their ODBs are mapped when computing the CRC. When the whole data-set has been read-back through the ICAP, the computed CRC is ready in the internal registers of the CRC32 module.

The CRC32 module consumes 125 slices, 233 LUTs and 32 flip-flops in the FPGA.

### 6.6.6 The Configuration Manager’s PicoBlaze

The ICAP PicoBlaze adjusts the configuration templates in the Bitstream BRAM and controls the ICAP driver based on the instructions issued by the scheduler’s PicoBlaze. As a result, it extends the basic functionality implemented by the ICAP driver and configuration

4 This polynomial is recommended by the IEEE, and is currently used in multiple telecommunication applications, including data link protocols such as HDLC, PPP, Ethernet and TCP/IP stack protocols like IP4, UDP and TCP.
templates to accomplish a set of low-level HWuK services which abstract the FPGA hardware. These services support most of common operations in RC, including: task allocation, reallocation, and deallocation, task context save and restore, as well as inter-task communications and synchronisation. Additionally, they allow for the management of the clocking resources included in modern FPGAs with branched clock-trees (e.g., Virtex-4) and support the fault-handling strategy presented in chapter 4. In fact, fault-handling is the only service which is periodically and autonomously executed by the PicoBlaze in low workload situations. The reliability of R3TOS depends on the configuration manager's PicoBlaze as it is responsible for (1) voting the results computed by the redundant instances of critical tasks, (2) detecting and reporting damaged resources to the allocator, and (3) scrubbing soft-errors in the FPGA's configuration memory. The set of the low-level R3TOS HWuK services implemented by the configuration manager's PicoBlaze are thoroughly described in chapter 7.

6.7 HWuK Internal Communications and Functioning

So far we have seen that HWuK is composed of three main components: the scheduler, allocator and configuration manager. The communication among these components is implemented by their respective PicoBlazes by using three 8-bit buses: Command bus (CMD), Acknowledge bus (ACK) and Parameter bus (PARAM). The CMD bus is used by the communication master (i.e., the scheduler's PicoBlaze), to trigger a specific operation in a slave (i.e., allocator or configuration manager), or to identify which type of data is available in the PARAM bus. The slave responds to any query from the master by issuing back the received CMD value through the ACK bus. This is done as soon as the data transmitted by the master has been received or when the operation invoked by the master has been finished. A zero value in the CMD and ACK buses means there is no operation in process; i.e., idle state. As some operations result in data to be passed from the slave to the allocator, e.g., the \((x,y)\) allocation of a task, conceptually the PARAM bus is bi-directional. However, physically, this bus is implemented as two separate uni-directional buses.

The main concepts related to the interaction of HWuK components when scheduling, allocating and executing a hardware task are illustrated in Figure 6.33. Note that in this example, it is assumed that the task to be executed is critical and it is only one row high. As can be seen in this figure, each data to be transmitted between scheduler-allocator and
scheduler-configuration manager has a different CMD identifier. Furthermore, some identifiers are not shown in this figure. For instance, only one word of clocking configuration is shown (i.e., CMD ID=10, 17 and 24, in scheduler-allocator communications, and CMD ID=14, 21 and 28, in scheduler-configuration manager communications) as the task is only one row high, but up to 6 identifiers are reserved to describe the clocking configuration of the highest tasks. When no CMD ID is specified, it is assumed the CMD bus is idle.

The communication handshake described in this section is amenable to diagnosis as any malfunctioning of the communicating components either leads to a bad CMD-ACK sequence or results in excessive delay in the response of the slaves.

6.7.1 Communication Monitor

The communication monitor is a hardware module which sniffs the CMD-ACK sequence to detect any malfunction. In order to do so, it evaluates the following four rules:

1. Any new operation must start when the CMD bus is idle, and once the operation is completed, the CMD bus must return to idle state again.

2. While executing an operation, the issued values on the CMD bus must be consecutive, except when alter_seq signal is enabled. For instance, in Figure 6.34, the alter_seq signal is enabled only when issuing command $M$ after having issued command $N+1$ in the previous cycle.

3. Prior to issuing a new value on the CMD bus, the previous value must be acknowledged on the ACK bus.

4. The longest delay in the response from a slave through the ACK bus is configurable by the designer.

If any of the rules above is violated, recovery actions are started in the HWuK (see chapter 4). Figure 6.35 shows the state diagram of the communication monitor, which is very simple, thus minimising the possibility of being affected by an error. It requires only 50 slices, 94 LUTs and 44 flip-flops, i.e., approximately half of the resources required by a PicoBlaze processor.
Figure 6.33: Simplified example of communications among HWUK components when scheduling, allocating, and executing a hardware task.
6.7.2 R3TOS Implementation Key Points

The key points of R3TOS functioning are listed below:

- Most of R3TOS processes are executed as optimised software routines in separate Xilinx tiny PicoBlaze processors. As a result, functioning updates can be easily made and the amount of consumed resources in the FPGA (i.e., area overheads), are kept within reasonable bounds. Only selected processes are executed in hardware (e.g., EAD updating).

- Each PicoBlaze deals with a different problem in R3TOS. For instance, while the configuration manager directly deals with FPGAs technology (e.g., configuration frames), the allocator manages the FPGA as a 2-D grid. As a result the HWuK is flexible and extensible as only selected components must be changed or added to adapt to new applications. For instance, a change in the reconfigurable technology requires only configuration manager be updated, while the allocator can continue to be the same. Moreover, this architecture allows for an easier maintenance.
In order to reduce time overheads, most of R3TOS processes are parallelised as shown in Figure 6.36.

**Scheduler**

**Allocator**

**Configuration Manager**

Figure 6.36: Parallelisation of R3TOS processes in the HWuK

### 6.8 Chapter Conclusion

This chapter thoroughly described the hardware implementation of the R3TOS HWuK, as well as its basic functioning. The HWuK is structured around three PicoBlazes. Two of them execute the task scheduling and allocation algorithms, respectively, while the other abstracts the complex particularities of the FPGA technology, by controlling access to the ICAP. Furthermore, each PicoBlaze is equipped with a dedicated BRAM where it stores its meaningful information. While the scheduler keeps track of the state of the tasks, sorting them into different task state queues, the allocator maintains the state of the FPGA area updated, and the configuration manager stores the configuration information to be transmitted to the FPGA's configuration logic. Finally, specific circuitry is included in the HWuK to speed-up the execution of selected time-demanding operations, and to detect and prevent any malfunctioning.

The next chapter describes the low-level services which are built upon this hardware and serve as the substrate to develop R3TOS-based reconfigurable applications.
Chapter 7

Hardware Abstraction Layer:
Low-Level R3TOS HWuK Services

Despite the fact that a R3TOS standard user is likely to interact with the FPGA hardware through a high-level software POSIX-like API, it might happen that in some specific applications he/she would need to take a low-level hands-on approach to meet strict real-time requirements, or simply to save FPGA resources or energy if there is no need to use a main CPU. In this case, when the application is to be based on bare hardware, a more agile interface to the FPGA resources is required in light of achieving improved performance and higher reliability levels. With this objective, R3TOS enables direct access to its Hardware Abstraction Layer (HAL). The HAL includes the set of low-level basic services implemented by the configuration manager, and hence, can be considered “the skin of the hardware” itself. An itemisation of the functions that are triggered in the configuration manager when executing each HAL service is shown in Table 7.1.

In general, the HAL turns the vast and complex FPGA hardware into an easy-to-use computing resource which can operate in connection with any CPU / software OS required by the developer. Note that the HAL lies beneath the scheduling and allocating services, which indeed do not account for most of the particularities of the underlying reconfigurable hardware (e.g., configuration frames). Besides, the HAL also deals with the particularities derived from the way R3TOS is conceived and implemented (e.g., management of data traces).
Chapter 7 - Hardware Abstraction Layer: Low-Level R3TOS HWuK Services

<table>
<thead>
<tr>
<th>Service</th>
<th>PBR</th>
<th>RBF</th>
<th>WMF2S</th>
<th>WSF2M</th>
<th>WSF2S</th>
<th>BIF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task (Re)Allocation Service</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Task Deallocation Service</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Task Preemption Service</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inter-task communication Service</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clocking Management Service</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Error-Handling Service</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

1 PBR: Partial Bitstream Relocation
2 RBF: Read-Back Frame
3 WMF2S: Write Multiple Frames to Single location
4 WSF2M: Write Single Frame to Multiple locations
5 WSF2S: Write Single Frame to Single location
6 BIF: Blank Frame

Table 7.1: Mapping of the R3TOS services to functions executed in the configuration manager

This chapter is aimed at grouping together all of the concepts related to the HAL services which have already been outlined in previous chapters as well as providing new explanations to complete a comprehensive understating of them. We acknowledge that although all the functionality implemented by these services has been tested separately, parts of it have not been integrated in the R3TOS complete system yet. In addition, we acknowledge that we could not test the effectiveness of the fault-handling service to localise permanent damage on the chip as we were unable to get any Xilinx part which had not passed the quality inspection (see section 7.6).

7.1 Task (Re)Allocation Service

The task allocation service is based on the Partial Bitstream Relocation (PBR) function implemented by the configuration manager. It allows to allocate a task (with TaskID=i) in a target location (X,Y) and running at a specific f_{CLK,i} clock frequency. While the location and the clock source of the task are decided by the allocator in order to improve efficiency, performance and reliability, the configuration manager is responsible for uploading the task’s partial bitstream to the corresponding location in the FPGA’s configuration memory.

It must be noted that the clock source decision leads to two different types of clocking configuration information, namely the configuration related to the BUFRs (e.g., BUFR_DIVIDE parameter) to generate the appropriate clock signal for the task, and the configuration related to the regional clock lines to deliver the generated clock signal to all of the sequen-
Chapter 7 - Hardware Abstraction Layer: Low-Level R3TOS HWuK Services

tial components inside the task. The former configuration information is managed by the
clocking management service (see section 7.5) as it refers to the regional clocking resources
included in the R3TOS static infrastructure. On the other hand, the latter configuration in-
formation is to adjust the internal architecture of the task and hence, it is managed by the
task allocation service.

All relevant information to direct the allocation of the task, including \((X,Y)\) and the con-
figuration associated with the regional clock lines, is transmitted by the scheduler’s PicoBlaze
to the configuration manager’s PicoBlaze. As shown in Figure 7.1, the latter redirects this
information to the required modules in the configuration manager and coordinates them to
perform the required operations. Specifically, the configuration manager’s PicoBlaze deliv-
ers: (1) \((X,Y)\) and the configuration information related to the regional clock lines to the ICAP
driver; (2) the coordinates of the upper-left and bottom-right vertices of the FPGA region as-
signed to the task, \((x_{TUL}, y_{TUL})\) and \((x_{TDR}, y_{TDR})\), to the Configuration Guardian (CG); and
(3) the \text{TaskID} to the external memory controller. Based on the \text{taskID}, and using the
pointer table located in the lowest part of the external memory (see chapter 4), the memory
controller initiates the bulk of the partial bitstream associated to that task to the bitstream
BRAM. The ICAP driver initiates the transfer of the partial bitstream from that BRAM to the
FPGAs’s configuration memory after at least 41 clock cycles. Note that this delay is necessary
to allow accessing the frame words in inverse order (i.e., from the 41st to the 1st), when allo-
cating a task on the bottom half of the FPGA. As explained in chapter 6, the frame addresses
as well as the regional clock line connections are adjusted by the ICAP driver on-the-fly.

![Figure 7.1: Functioning of the task allocation service](image)

246
The management of redundancy when dealing with critical tasks is nearly transparent to the ICAP driver as the partial bitstreams of these tasks already include the necessary configuration commands to allocate up to three instances of them on the FPGA. The ICAP driver only needs to keep track of the locations associated with each task instance. As the redundancy is based on MFWR commands, the frame address sequence in the tasks’ bitstream is indeed the same as for the Write Multiple Frames to a Single location (WMF2S) function implemented by the ICAP driver: FAR1-FAR1-FAR2-FAR3 (see Figure 6.26).

Once the partial bitstream of a task is successfully uploaded to the FPGA's configuration memory, other HAL services are invoked to deallocate overlapping tasks (i.e., task deallocation service, see section 7.2), to deliver input data to the allocated task (i.e., inter-task communication service, see section 7.4), and to set-up the appropriate clock frequency for the task (i.e., clocking management service, see section 7.5). The Hardware Semaphore (HWS) of the task remains disabled while all of these services are executed (i.e., the task is in reset state). Afterwards, the Global Set/Reset (GSR) is activated through the STARTUP logic to force all registered signals in the task be loaded with their INIT values. Finally, the HWS is enabled and the task starts performing active computation. Note that in order to enable task’s HWS, a Read-Modify-Write sequence is executed where the Read-Back Frame (RBF) and either Write Single Frame to a Single location (WSF2S) or Write a Single Frame to Multiple locations (WSF2M) functions are consecutively invoked in the ICAP driver.

7.1.1 (Re)Allocation of Partial Bitstreams in-between Clock Regions

Despite of not being currently included in the standard task allocation service, R3TOS is also able to allocate tasks in-between different clock regions on the FPGA die. As far as we know, this is the first time this capability is reported.

Traditionally, the vertical relocatability of the hardware tasks has been constrained by the amount of clock regions in the FPGA. Indeed, as pointed out in chapter 2, the basic configuration information unit when relocating a hardware task is the configuration frame. While the frame addresses are modified based on the target location for the task, the frame data remain unchanged. As a result, the vertical placement for a relocatable hardware task within a clock region cannot be modified at runtime. The suitable way to deal with this limited flexibility has been to make the height of the tasks be a minimum number of FPGA
clock regions, as explained in chapter 5. This brings some benefits. First, the management of the FPGA resources is simpler as a single bit can be used to represent all of the resources included in an FPGA column (i.e., available or not available to be used). Besides, the hardware tasks span a minimum amount of clock regions, resulting in faster configuration times and best exploitation of FPGA resources.

In most situations the relocation solution described above is enough, especially when using large FPGAs. However, when a significant amount of device's resources are damaged, this solution turns ineffective. As shown in Figure 7.2, the tasks would be successfully allocated in-between the damaged resources as long as they could be vertically shifted, lying over FPGA's clock region boundaries. In other words, relocation should not be constrained by the logical scheme of FPGA's configuration memory, instead it should work at the finest granularity of the physical chip (i.e., CLBs, BRAMs or DSP48s). By doing so the overall relocatability would be improved by a factor of 16x for the tasks which include CLBs exclusively, by a factor of 8x for the tasks which also include DSP48s, and by a factor of 4x for the tasks which include BRAMs.

![Figure 7.2: Benefit of allocating the hardware tasks in-between clock regions](image)

In order to arbitrarily relocate a task in-between different clock regions, its configuration information is vertically shifted, being appropriately distributed along several vertical adjacent frames. This is shown in Figure 7.3, referring to the task $\theta_k$ in the original and relocated locations depicted in Figure 7.2. As explained in chapter 2, the configuration related to each
CLB spans 2.5 words (i.e., \( \frac{40}{16} \)), and the configuration related to each BRAM spans 10 words within each frame (i.e., \( \frac{40}{4} \)). Therefore, this is the basic unit of configuration information that is shifted when relocating the tasks. Usually it is more convenient to use a granularity of 2 CLBs in order to circumvent data shifting within each word, i.e., configuration data is moved in units of 5 entire words within the frames.

The only word that is not shifted is the 21st word, which configures the regional clock routing and also allocates the Frame_ECC codes. In fact, the regional clocking configuration is separately managed. Thanks to the highly adaptable clocking infrastructure of R3TOS, where the resources of the tasks are fed with dedicated clock signals that are generated in each of the clock regions (i.e., by BUFRs), it is indeed possible to dynamically enlarge the tasks along other frames, assigning to them FPGA resources which lie outside their original clock regions. The latter resources are fed with new clock signals which are conveniently set-up using the corresponding BUFRs. Note that the configured clock rate must be the same in all of the BUFRs which feed the same task. For instance, when relocating task \( \theta_k \) in Figure 7.3, the clocking configuration in clock regions A and B remain the same in the original and target locations, but a new clock signal must be set-up in the clock region C to feed the resources assigned to \( \theta_k \) in that region.

Note that allocating a task in-between clock regions requires the capability to compute new Frame_ECC codes on-the-fly as the content data of the frames change. For instance, in Figure 7.3, the Frame_ECC codes included in the clock region C are not used in the original location of the task \( \theta_k \), while they are used in the target location. The Frame_ECC codes are computed by the configuration manager’s PicoBlaze based on the polynomial presented in chapter 2 (see Table 2.10).

Some remarks are in order here. First, the fact that the configuration of a task spans more frames than strictly necessary results in a longer configuration time and less efficient use of FPGA resources. Indeed, the configuration frames cannot be shared between different tasks unless the clock of the firstly allocated task is temporarily stopped while allocating the second task. Otherwise the user information of the former task might be corrupted when writing the configuration of the latter task, or when issuing the GSR signal to initialise its flip-flops. Second, the Empty Area Descriptor (EAD) and the area matrices managed by the allocation algorithms should use the same granularity as the relocation method, i.e., they
Figure 7.3: Relocating $\theta_k$ in-between clock regions

should indicate the state of each individual FPGA resource. This leads to higher storage resource needs and longer times for making the allocation decisions.

Currently no specific hardware support for inter-clock region relocation is included in the R3TOS HWuK. In fact, this capability has been tested on a real Virtex-4 FPGA using exclusively software routines which run on the configuration manager’s PicoBlaze. The operation sequence to perform inter-clock region relocation has been the following. First, the task is allocated using the standard allocation service on a location where the immediately above clock region is empty. Then, the task is vertically shifted in such a way that it uses the resources included in the bottom part of the above region. In order to do so, task’s frames are individually read-back from their original location, temporarily buffered and combined to give raise to new frames that are written-back to the target location. Note that these frames include renewed Frame_ECC codes which account for the new configuration information they include. Afterwards, the necessary clock signals are set-up in the BUFRs. Note that there is no need to adjust the task’s internal connections to the regional clock lines as the same BUFRs are used in all of the clock regions that the task spans. Finally, the HWS of the task is enabled and it starts performing computation in the target location.
7.2 Task Deallocation Service

Task deallocation is necessary in order to avoid routing conflicts when a task $\theta_i$ is allocated on a location that overlaps with the footprint of a previously executed task $\theta_j$. Note that in this situation, some of the resources assigned to $\theta_i$ are likely to be connected to other resources located out of its boundaries, but previously contained within the boundaries of $\theta_j$ (see Figure 7.4). These undesired connections might interfere with the normal operation of $\theta_i$, being a potential threat for the correctness of computation. In order to remove the latter connections, the configuration information of $\theta_j$ which has not been overwritten when loading the bitstream of $\theta_i$ must be blanked in the FPGA's configuration memory prior to enabling $\theta_i$'s HWS. However, the blanking operation affects neither BRAM content frames nor LUT content frames as the data traces must still remain in the FPGA after deallocating the producer tasks. Data traces are blanked when all of the expected consumer tasks have already accessed them.

![Figure 7.4: Allocating $\theta_i$ within the footprint of a previously executed task $\theta_j$](image)

The task deallocation service is mainly based on the Blank Frames (BlF) function implemented by the configuration manager. While the BlF function writes all-zeros to all of the (minor) frames included in a specific FPGA column, the task deallocation service gives the necessary support to blank a given rectangular region within the FPGA surface. Hence, the BlF function is repeatedly executed with the appropriate frame address and amount of minor frames in each consecutive resource column within the region to be blanked. Since the latter information is obtained from the FPGA descriptor which is held in the lowest part of the Bitstream BRAM (see chapter 6), the only input parameters to this service are the co-
ordinates of the two corners of the region to be blanked (i.e., upper-left and bottom-right corners). Depending on the way the allocated task overlaps with the previously executed task, several rectangular regions must be blanked in the FPGA. For instance, in Figure 7.4, up to four different rectangular regions (A, B, C, and D) must be blanked after having allocated the task $\theta_i$ within the footprint of the previously executed task $\theta_j$. The rectangular regions to be blanked are delimited and then transmitted by the scheduler’s PicoBlaze to the configuration manager’s PicoBlaze.

Figure 7.5 shows the HWuK circuitry that supports the task deallocation service as well as the operation sequence. The only significant remark here is that the configuration manager must enable the writing on the region to be blanked prior to effectively performing this operation. Indeed, the upper-left and bottom-right corners of the region to be blanked are equal to the $(x_{TUL}, y_{TUL})$ and $(x_{TDR}, y_{TDR})$ coordinates to be written in the CG.

![Figure 7.5: Functioning of the task deallocation service](image)

7.3 **Task Preemption Service: Context Save and Restore**

As described in chapter 4, preemption is not currently allowed in the execution phase of the tasks. Instead, the duration of the execution phase is limited and usually known, i.e., the amount of time needed by pure hardware to come up with the results can usually be predicted with precision. As a result, there are a set of pre-defined instants when the tasks finish a partial computation and the results computed by them can be accessed. If the tasks are reentrant, accessing the partial results after each execution is conceptually similar to a context save as the execution of these tasks can be later resumed by delivering the partial
results again to them (i.e., by restoring tasks’ context). Unlike in a spontaneous task pre-
emption, where the state of all of the sequential components included in the task must be 
saved (see Figure 7.6a), the amount of results to be saved in R3TOS is reduced and localised 
in specific configuration frames, being conceptually limited to the data stored in the data 
buffers of the tasks (see Figure 7.6b).

Moreover, since these buffers are specifically conceived for exchanging data, they are 
implemented using either LUT-RAM/ROMs or BRAMs, enabling access to data directly in 
the FPGAs configuration memory (see chapter 2). However, when the implementation of 
the hardware tasks is not up to the application developer (e.g., the tasks are developed by a 
third party company or a high-level design tool which does not allow to specify the imple-
mentation details is used), it might happen that part of the data to be exchanged is mapped 
to flip-flops or to disjoint, or even unknown, resources within the tasks. If this is the case, 
there is a need to save the state of all of the resources which could potentially store user in-
formation. This includes flip-flops, whose content can be only accessed using GRESTORE 
and GCAPTURE commands.

![Figure 7.6: Task context](image)

Despite the fact that task preemption is not envisaged in the current implementation of 
R3TOS (based on a Virtex-4 FPGA) due to the time overheads it implies\(^1\), this mechanism 
could be useful in the future, if reconfiguration speed of FPGAs folds up, as it might be\(^2\). 
Indeed, task preemption has been successfully used for several decades in the software field,

\(^1\)Preempted tasks must be configured twice.
\(^2\)While the highest achieved reconfiguration speed in a Virtex-4 is 576 MB/s [Shelburne et al., 2008], a reconfiguration speed as high as 2.2 GB/s is reported for new FPGA families (e.g., Virtex-5) [Hansen et al., 2011].
where the time penalties due to context save and restore are not significant. In this thesis, the necessity for a preemption mechanism in a real-world application is shown in the scope of our Software Defined Radio (SDR) prototype described in chapter 9. Besides, saving the context of a task is necessary for implementing checkpoint-rollback recovery.

The operation sequence to save the context of a task is as follows. First, the clock of the task is stopped so that computation is interrupted. Then, a GCAPTURE command is issued to make the content of flip-flops accessible in the FPGA’s configuration memory. Finally, task’s configuration information is read-back using the RBF function implemented by the configuration manager. At this point, note that the read-back information includes both exclusively configuration data (i.e., task’s partial bitstream) and user data (i.e., task’s context). Since the read-back information is likely to be large, it should be temporarily stored in the external memory. In some cases (e.g., when dealing with reentrant tasks), this information could overwrite the original task’s partial bitstream stored in the latter memory. If this is the case, the task allocation service presented in section 7.1 can be used to restore the task context. Hence, the state of BRAMs, SRL16s and LUT-RAM/ROM is automatically restored when loading the saved context without requiring any other action to be taken, and the state of flip-flops is restored as a result of executing the GRESTORE command. This is originally intended to initialise the flip-flops with the reset values, but in this case, it loads to the flip-flops the previously read-back user data which indeed overwrites the original INIT values.

It is important to note that the task preemption service assumes that writing to all flip-flops in the FPGA is disabled at start-up. Indeed, writing is only enabled for a set of particular flip-flops prior to restoring the context of a task, and it is disabled when finishing, prior to enabling the HWS of that task.

Despite the fact that the task preemption service described in this section has been tested on a real Virtex-4 FPGA with small hardware tasks, the current implementation of the R3TOS HWWK includes limited support for it. Specifically, the external memory controller is not currently capable to temporarily save tasks’ context in the external memory. Besides, it should be convenient to include a specific configuration template in the Bitstream BRAM to give support to the process of enabling and disabling the writing to the flip-flops associated
to the task whose context is to be restored (i.e., to manage the mask register associated to these flip-flops).

Finally, I acknowledge this service has been mainly implemented by Mr Raúl Torrego [Torrego et al., 2012b]. I participated in its development as an advisor.

### 7.4 Inter-task Communication Service

In general, the inter-task communication service is aimed at providing support for reallocation of the content of the Output Data Buffer (ODB) of a producer task \( \theta_i \) to the Input Data Buffer (IDB) of a consumer task \( \theta_j \). So far we have seen that this operation can be conducted in various ways in R3TOS, requiring different functions be triggered in the configuration manager in each case.

The most generic communication method is based on (1) the RBF function, to read-back the frames associated to the ODB of the data producer task, and either on (2) the WMF2S function, when dealing with non-critical consumer tasks, or on (3) the WSF2M function, when dealing with triplicated critical consumer tasks. The upper-left and bottom-right corner coordinates of the data consumer task are conveniently written to the CG prior to executing the WMF2S or WSF2M functions. Unlike when saving the context of a task for an undetermined amount of time, when carrying out inter-task communications, the exchanged data is read-back and immediately written to the target location, using the bitstream BRAM to temporarily buffer it.

The inter-task communication service envisages the possibility of exchanging data among buffers implemented using different FPGA resources (see Figure 7.7). This could be useful when communicating tasks which have not been developed according to the R3TOS guidelines (e.g., using flip-flops to store data) with tasks that do follow the R3TOS recommendations, and thus use exclusively LUTs or BRAMS. Moreover, it permits to exploit often existing extra memory positions in BRAMs to hold small sets of data, which are stored in LUTs in the other communicating task. The latter situation can be found when communicating High Bandwidth Communication (HBC) tasks with Low Bandwidth Communication (LBC) tasks.

In order to adapt the format of the data to the needs of the different FPGA resources, the read-back frames need to be processed in the Bitstream BRAM, resulting in longer com-
munication overheads. As pointed out in chapter 2, LUTs and flip-flops store their content “in clear”, i.e., the 16 data bits stored in a LUT occupy consecutive positions within the bit-stream, but the BRAM content data is distributed along a set of frames in an a non-trivial way. We only know that 256 consecutive data bits and 32 consecutive parity bits are mapped to the same BRAM content frame.

In light of this, exchanging data between LUTs and flip-flops is trivial: the data is extracted from the original location within the read-back frames and then relocated to the appropriate target location within the frames to be written. This means that 16 flip-flops are grouped together to form the content of a LUT (see Figure 7.7a) and, on the contrary, each of the 16 bits of a LUT are copied to the INIT bits corresponding to separate flip-flops. Note that GCAPTURE and GRESTORE commands are issued when needed.

However, exchanging data between LUTs or flip-flops and BRAMs is more complicated as this requires to deal with the “obscured” data format used to store the BRAM content in the bitstream. The easiest and fastest way to deal with this issue is to use a coding-decoding BRAM (e.g., the bitstream BRAM). As shown in Figure 7.7b, flip-flop or LUT data is extracted from the original location within the read-back frames and written “in clear” to 256-bit length memory segments within the bitstream BRAM. Note that the data stored in the LUTs contained in a column of CLBs (i.e., 2 Kb) fits in 8 BRAM data segments. Afterwards, the memory segments are read-back through the ICAP and finally written to the target BRAMs, i.e., to the IDB of the data consumer task. The latter access through the ICAP is necessary to convert the data “in clear” to the suitable format demanded by the BRAMs. In order to exchange data in the opposite way (see Figure 7.7c), content frames are read-back from the source BRAM, i.e., data producer’s ODB, and written again to frames which correspond with different memory segments within the bitstream BRAM. Note that it is then possible to access data “in clear” through the latter BRAM’s ports. Finally data is rearranged to the suitable location within the frames prior to being copied to the LUTs or flip-flops.

The remaining two communication methods are mainly based on the PBR function implemented by the ICAP driver. The first method consists in allocating the data consumer task in such a way that it has direct access to the BRAMs where input data is stored. As explained in chapter 4, the partial bitstreams of the tasks are parsed at design time in order to remove the BRAM content frames associated to the data buffers. As the BRAM intercon-
Figure 7.7: Data exchange between buffers implemented using different FPGA resources (Stripped arrows represent operations performed through the ICAP)
nection frames are not removed, the consumer task gains access to the BRAMs when it is allocated on the FPGA (i.e., previous interconnection information is overwritten), but the content of the BRAMs remains the same.

The second method requires the allocation of a Data Relocation Task (DRT) prior to configuring the data consumer task. The DRT moves the data to a different set of BRAMs, which can be directly accessed by the data consumer task when the latter is finally allocated. The inter-task communication service automatically manages the DRTs, selecting the appropriate one for each communication situation, retrieving its associated partial bitstream from the external memory, and once it is uploaded to the FPGA’s configuration memory, invoking the clocking management and the task deallocation services as needed. Indeed, as explained in previous chapters, the DRTs are treated as standard computing tasks: they are provided with the necessary clock signals, and the allocated tasks which overlap with them are deallocated. A detailed description of the low-level technology implications of DRTs is provided in chapter 6.

7.5 Clocking Management Service

While the general clocking infrastructure used in R3TOS has been described in chapter 6, this section is aimed at describing the way this infrastructure is adapted to deal with the different clock requirements imposed by the tasks, to cope with occurring faults in the clock-tree as well as to keep the power consumption as low as possible.

First of all, we note that, unlike in related work, in R3TOS each task receives as many clock signals as clock regions it spans. While all of the clock signals delivered to a task must be of the same frequency, they can be routed through different regional clock nets in each clock region (see Figure 7.8). This depends on the occupation of the FPGA as well as on any existing damage in the clock-tree, and requires the capability to route the clock signals inside the tasks at runtime. Indeed, R3TOS implements a complete online clock routing mechanism, where some parts are autonomously managed based on reliability premises (e.g., BUFGCTRLs), and the others are managed by either the clocking management service (e.g., BUFRs and RCMUX) or by the aforementioned task allocation service (e.g., clock routing inside the tasks). So far we have seen the central role this mechanism plays for inter-clock region task relocation.
In contrast, in the related work there is a single clock source per task, mainly due to implementation constraints. The vast majority of reported reconfigurable systems do not use regional clocking resources, and most of the systems which do use them are not capable of routing the regional clock signals inside the tasks (e.g., [Jara-Berrocal and Gordon-Ross, 2010]). As pointed out in chapter 3, the latter approaches include the BUFRs as a component of the tasks and therefore, the clock routing remains unchangeable inside them.

The clocking management service controls thus two components: BUFRs and RCMUXes.

The process of setting-up the clock sources in the BUFRs basically consists in writing the appropriate BUFR_DIVIDE parameters in order to generate the required clock rate for the tasks. It is important to note that all of the clock sources are synchronised because they are derived from the same (global) clock signal, which is the input to the BUFRs. When only one frequency is needed in a clock region, the other BUFR is disabled in order to reduce the power consumption. In this functioning we note the central role of HWS. While the clocking adjustments are sequentially performed (i.e., only one frame can be read or written through the ICAP at a time), the HWS allows to enable all of the clock signals which feed the same task at a time, circumventing any potential synchronisation problem. For simplicity and predictability the clock signals are not modified during the execution phase of the tasks, neither their rate nor their routing.
When one of the BUFRs in a clock region is damaged, the RCMUX is used to switch a spare clock signal from any of the 4 BUFRs located in the top or down neighbour clock regions. Hence, as shown in Figure 7.9b, when any of the hardware task spans more than one clock region in height, the non-damaged BUFRs of these regions can be used in the regions with damaged BUFRs. Besides, as shown in Figure 7.9c, one of the BUFRs in a region where only one clock frequency is needed can feed the tasks placed in the adjacent clock regions with damaged BUFRs. Two remarks are in order in Figure 7.9. First, BUFRDC refers to the BUFR Diagnostic Circuit explained in chapter 6, which allows to detect damaged BUFRs. Second, the RCMUX is not a physical FPGA resource, it is implemented by appropriately configuring the Programmable Interconnection Points (PIPs) which select the input connection of the regional clock lines. We posit that this novel use of the branched clock-tree available in modern Xilinx FPGAs significantly increases the reliability of the system.

Finally, note that despite being more complex to implement, the same strategy could be used to switch a spare BUFGCTRL when one of the input clock sources to the active one fails; i.e., the system will fail if the remaining clock source also fails.

### 7.6 Fault-Handling Service

The general fault-handling strategy used in R3TOS has been outlined in chapter 4. There is little more to be said about the way R3TOS deals with HWuK functioning failures: the system is restarted by completely reconfiguring the FPGA. On the other hand, as shown in Figure 7.10, R3TOS envisions two different diagnostic tests to be carried out in response to kernel configuration errors and to task computation errors, respectively. While the former is not intrusive to allow R3TOS to continue working normally, the latter is intrusive in light of achieving more precise diagnosis results.

As shown in Figure 7.10a, R3TOS relies on the Frame_ECC codes included in the configuration frames to periodically detect kernel configuration errors as well as to correct single soft-errors, if any. Indeed, the upset bit in a frame is identified by means of the Frame_ECC syndrome, inverted and written back to the configuration memory; i.e., the flipped bit is scrubbed on the configuration memory. Afterwards, the frame is read-back again to check whether the error continues to exist. If so, permanent damage is assumed to be the source of that error and the system initiates a fail-safe shutdown. It is important to note that we have
Chapter 7 - Hardware Abstraction Layer: Low-Level R3TOS HWuK Services

**Figure 7.9:** Replacing damaged BUFRs...

(a) Normal configuration

(b) ...with spare resources in large tasks spanning regions

(c) ...with unused resources in regions where only one clock frequency is needed
amended the existing decoupling problem between the ICAP and the Frame_ECC logic in Virtex-4 FPGAs, as described in chapter 6.

In R3TOS, computation errors are detected when any of the three redundant instances of a critical task computes a different set of results (i.e., value domain errors), or when the HWS of a task that should have finished its computation does not indicate so (i.e., time domain errors). R3TOS diagnoses the source of computation errors and, in case they are due to permanent damage on the chip, it prevents the future use of the damaged resources. The diagnostic mechanism implemented in this thesis harnesses the ICAP to test the functionality of the FPGA’s configuration memory. Indeed, a configuration bit can be corrupted because either the memory cell where it is held has been hit by an ionising particle, i.e., the configuration bit is upset but the memory cell is not damaged, or as a result of existing permanent damage around the memory cell. As the configuration cells are distributed along the entire
FPGA die, any damage on the chip is likely to affect some of them. Additionally, the source of computation errors can be damage in the clocking infrastructure. These errors are diagnosed by checking all the diagnostic circuits associated to the BUFRs that deliver clock signal to the erroneous task instance.

Before executing the diagnostic test, the FPGA Region Under Test (RUT) is blanked and the clock delivered to that region stopped (see Figure 7.10b). This ensures the resources included in the RUT are not operating, preventing any potential interference with the diagnostic test execution. Note that the RUT exactly matches the region where the hardware task instance that computed a wrong set of results was allocated. Unlike in the standard task deallocation service, the blanking operation carried out on the RUT also clears user modifiable information. The diagnostic test first checks that all of the configuration bits associated with the resources included in the RUT have been loaded with zeros, and then, after configuring all-ones frames in the RUT, that they are truly set to logic level '1'. As for blanking, all-ones loading operation also applies to user modifiable information. In order to speed up the execution of the test, both CRC32 Module and Frame_ECC primitive are used when reading-back the frames. The former checks the correctness of user writable resources (e.g., SRL16s, BRAMs and flip-flops), while the latter covers fixed resources (e.g., LUTs that implement a logic function and switch matrices). If either the CRC32 or Frame_ECC values are not the expected ones, the corresponding frames are thoroughly analysed to localise the stuck-at bit(s) and the associated resource(s) are marked as not available in the FPGA State BRAM (i.e., damaged).

In order to circumvent harmful FPGA configurations while performing the diagnostic test, only safe guaranteed configuration patterns are used. Namely, with the objective of avoiding shortcuts, all-one frames are loaded in two times. First, LUTs and flip-flops are kept to zero state and routing configuration bits are set to one, with the exception of the PIPs that route Vdd power lines in the TIEOFF cells. Then, routing bits are cleared and LUTs and flip-flops are set to one. Besides, only one frame is allowed to be configured with all-ones at any time. Hence, prior to switching to diagnose a new frame, the previously analysed frame is loaded with all-zeros again. In general, our diagnostic test is dependent on the bitstream format of the FPGA, and its fault coverage is limited. For instance, it does not cover the TIEOFF cell's PIPs.
In light of the above limitations, we note that the diagnostic mechanism to be adopted in the final implementation of R3TOS is still open. For instance, the RUT could be loaded with a Built-In-Self-Test (BIST) to carry out an exhaustive test of all routing wires and logic resources [Abramovici et al., 2004, Smith et al., 2006, Dutt et al., 2008, Amouri and Tahoori, 2011]. Besides reducing the occupation of the ICAP, BIST circuits are claimed to allow for a more effective and faster diagnosis with higher fault coverage. On the other hand, the major weakness of BIST circuits is the risk of malfunction when sensitive parts of them (e.g., test pattern generator or output response analyser) are mapped to damaged resources. We note that reliability is precisely one of the main advantages of the ICAP-based diagnosis as it is exclusively based on the presumably highly-reliable R3TOS HWuK. The other weakness of BIST is the need to create a vast battery of diagnostic circuits to cover the wide range of sizes and shapes of RUTs. Despite the BIST research line has not been explored in this thesis, any work previously done in this field is potentially amenable to be included in R3TOS. The only requirement imposed by R3TOS is the necessity to access the diagnosis results in the BIST circuit remotely through the ICAP, i.e., the results should be stored in either LUTs or BRAMs.

7.7 Chapter Conclusion

This chapter explained the HAL services available in R3TOS, which are mainly implemented by the configuration manager. These are aimed at providing support to the user for carrying out the most common operations in RC, including task allocation, deallocation, preemption, synchronisation, and inter-task communications. In addition, the HAL services ease the integration of the advanced capabilities delivered by reconfigurable hardware into a traditional processor/OS platform. A description of this integration is provided in chapter 8.

Besides describing standard operations in RC, the chapter presented two novel capabilities of partially and dynamically reconfigurable FPGAs: (1) inter-clock region task reallocation of partial bitstreams and (2) online routing of regional clock signals. The former capability permits to change the vertical allocation of tasks in the FPGA clock regions, making it possible to circumvent permanent damage on the chip at the finest resource granularity. The latter capability, which is indeed necessary to support inter-clock region task reallocation, allows to feed each task with a different clock frequency as well as to recover from failed clock sources.
R3TOS Main CPU and Application Programming Interface

Our R3TOS solution is completed with a POSIX-like Application Programming Interface (API), which provides high-level software-centric users with a familiar way to access the low-level services implemented by the HWuK, i.e., Hardware Abstraction Layer (HAL), and ultimately to exploit the FPGA resources. The HAL is thus wrapped with a software OS layer, called software microkernel (SWuK), which is executed on the main CPU.

The OS-CPU duple provides the basic platform to execute application software routines, i.e., the serial portions of the application which are not amenable to be accelerated by parallel processing or by computation specialisation. On the whole, the combination of R3TOS HWuK and SWuK results in a good framework to develop hardware-software hybrid applications. However, the achieved higher abstraction level is traded off with a loss in performance and an increase in power consumption, as the CPU needs to execute extra OS processes. Besides, the system becomes more complex, and thus more error prone.

This chapter presents both the SWuK API and the main CPU upon which it is implemented. Additionally, it explains the way hardware tasks are represented and managed in the high-level software environment. We note from the very beginning that the currently implemented R3TOS SWuK is for demonstrative purposes and hence very basic (mostly based on a commercially available solution: FreeRTOS).
8.1 The R3TOS Main CPU

As mentioned in previous chapters, R3TOS HWuK is conceptually able to work in conjunction with most existing microprocessors, either on-chip or off-chip. Indeed, HWuK implements a generic interface with the main CPU, which is based on interruptions and shared memory (i.e., Task BRAM and input/output data buffers).

In the current implementation of R3TOS, a 32-bit Xilinx MicroBlaze is chosen as the main CPU. Notably, MicroBlaze is fully supported by Xilinx design tools and is provided as soft-core, allowing for possible customisation of its implementation on FPGA based on various requirements. For instance, MicroBlaze can be customised to fit in a target FPGA location, with a specific shape, amount and type of resources. This is especially interesting in the scope of R3TOS, where the objective is to place the static circuitry in such a way that the amount and length of the static routes across the chip is minimised, thus increasing the allocatability of hardware tasks. Furthermore, Xilinx Inc. has recently provided support for using ECC protected BRAMs to store the program code that is executed by a set of redundant MicroBlaze cores [Xilinx Inc., 2011a, Hardcastle, 2012], thus closing the gap with other processors traditionally designed for reliability (e.g., Gaisler’s LEON3-FT [Aeroflex Gaisler Inc., 2012]). Indeed, we note that the full R3TOS implementation could use ECC-protected BRAMs as we have enabled the PicoBlaze processors used in the HWuK to run from these memories (see Appendix A).

We will not go into implementation details of the MicroBlaze-based main CPU as we have mostly used standard peripherals and design tools provided by Xilinx Inc., i.e., Xilinx Platform Studio (XPS). The CPU includes the expected functionality by an application developer, including networking capabilities (e.g., Ethernet and RS232), high-performance support (e.g., DMA), and real-time support (e.g., timer). We will note only three remarks. First, a memory access arbiter has been developed to circumvent collisions when accessing the single large capacity external memory, which is shared between the main CPU and HWuK. Indeed, the memory stores both the partial bitstreams associated with the hardware tasks and the code of the software tasks, as explained in chapter 4. In order to increase performance, the MicroBlaze is equipped with a cache memory implemented using on-chip BRAMs, where the SWuK code lies. Second, the main CPU is coupled with BRAM and LUT
Chapter 8- R3TOS Main CPU and Application Programming Interface

based Input and Output Data Buffers (IDB and ODB) to enable data exchange between software and hardware tasks. Finally, a direct link to the Task BRAM and interruption signals coming from the HWuK is provided. A block diagram depicting the components through which the main CPU and HWuK interact is shown in Figure 8.1.

![Block Diagram](image)

**Figure 8.1:** Main CPU: block diagram

### 8.1.1 External Memory Access Arbiter

The memory arbiter receives access requests from both the HWuK (HWuK_req) and the MicroBlaze (MB_req), and grants access to only one of them at any time, by enabling either HWuK_ack or MB_ack signals, respectively (see Figure 8.2). We note that we have made slight modifications in the standard memory controller provided by Xilinx Inc. to make it compatible with our memory access arbiter. Specifically, an access grant waiting state has been added to the FSM. In this state the access request signal is activated and the acknowledge signal activation is awaited. As the latter state is implemented in hardware, the time spent in it results in “idle” time in software, and in case it spans too long, it might eventually lead to a “stuck” assumption by the SWuK. To avoid this, the SWuK keeps track of the memory state at all times by means of a semaphore, i.e., occupied by HWuK or available. The interruptions to signal memory accesses by HWuK are generated by the memory arbiter. In case the memory is occupied, SWuK avoids accessing it and switches to execute code that is cached in the on-chip cache memory. Note that this could be a potential way to partially hide the configuration phases of the hardware tasks.
8.1.2 HWuK-SWuK Interface

Data Interface: Input Data Buffer and Output Data Buffer

The data buffers connected to the main CPU are exactly the same as explained in chapter 6. The only significant difference is that they are accessed through the system bus: data is copied to the ODB and read from the IDB. Note that the read data from the IDB, previously generated by hardware tasks, can then be copied either to the data segment of the corresponding consumer software task in the external memory, or left in the cache for immediate use. Analogously, the written data to the ODB, which will be processed by hardware tasks, can be retrieved either from the data segment of the corresponding producer software task in the external memory, or directly from the cache memory.

An unresolved problem is how to deal with the different types of IDBs and ODBs of the tasks. Ideally, a single and universal data buffer should be included in the main CPU, which should be able to adapt to the requirements of the different communicating hardware tasks. However, this is currently done by including a different data buffer for each type of buffer used in the hardware tasks.

Control Interface: Task BRAM and Interrupts

One of the ports of the Task BRAM in the HWuK is connected to the MicroBlaze, which accesses it through a standard BRAM controller connected to the system bus. Despite the fact that the whole BRAM is accessible from the main CPU, only three 32-bit dedicated posi-
tions are used to exchange data with the HWuK, as explained in chapter 6. This minimises the probability of corrupting the memory content by the main CPU. However, for debugging purposes it is still useful to access the whole memory content from the main CPU. Two of the dedicated 32-bit memory positions are used to exchange data from the main CPU to the HWuK, and the other is used in the opposite direction. Every time memory is written to by one of the parts (i.e., SWuK or HWuK), an interruption is provoked in the other part, so that it can read the written information. When this occurs, the interruption is cleared. The most significant 8-bits in the shared memory positions are used to exchange control commands, while the remaining bits are used to exchange 8-bit parameters, including task's ID, data retrieval time, execution deadline, execution time, data delivery time and data producer task ID. As in the case of HWuK component interaction, every control commands is acknowledged when the required operations are finished (see Figure 8.3). Communications can be initiated either by SWuK, when marking a task as ready, or by HWuK, either when indicating that a task has been scheduled and requires input data be delivered, or when a hardware task has completed its computation and results are ready in the IDB. In the latter two situations, the passed parameter from HWuK is the TaskID of the scheduled task or of the task that has completed its computation, respectively.

8.2 The R3TOS Software Microkernel (SWuK)

The R3TOS SWuK is currently based on FreeRTOS [Richard Barry, 2011]. The latter is an easy-to-use and open source real-time microkernel specifically designed to have a small memory footprint. Indeed, our FreeRTOS porting to R3TOS requires 29.8 KB, thus fitting in only 16 BRAMs. Two features of FreeRTOS are especially attractive to us. First, its high dependability, which is supported by the fact that a microkernel derived from it (i.e., SafeRTOS), has been certified for safety-critical applications. Second, its popularity, which is confirmed by the 2011 EETimes embedded systems market study, where FreeRTOS comes in top in two categories: the kernel currently being used, and the kernel being considered for the next project to develop. This is precisely our objective: provide R3TOS with the most attractive software skin for application developers.

FreeRTOS implements message queues as well as binary, counting and recursive semaphores and mutexes for communication and synchronisation between real-time tasks, or
Figure 8.3: Simplified example of communications between SWuK and HWuK when executing a hardware task
between real-time tasks and interrupts. Note that these mechanisms are natively available for software tasks, and must be extended to be used with hardware tasks. Indeed, the interaction between software and hardware tasks occur as outlined in chapter 4, and better explained in section 8.2.1.

While the core part of FreeRTOS has been kept intact in our porting to R3TOS, new Interrupt Service Routines (ISRs) have been programmed to enable communication with the HWuK. Likewise, the scheduler included in the commercial distribution of FreeRTOS has been modified to provide the necessary support for dealing with hardware tasks, and new functionality has been developed to ease common operations in R3TOS (e.g., DMA transfers between IDB, ODB and external memory). Finally, we note that the kernel tick used in the SWuK, which is in the range of milliseconds, is coarser than that in the HWuK. This reflects the achievable less accurate time precision when using software instead of hardware.

8.2.1 Management of Hardware Tasks in SWuK

The application developer can indistinguishably use software or hardware tasks, as both of them are managed in a uniform way by R3TOS. The major difference between them is that, while the body of a software task includes the computation to be performed, hardware tasks have a “ghost software body” whose main objective is to make them manageable in SWuK (see Listing 8.1). Nevertheless, we note that although the generic software body of a pure hardware task, which is shown in Listing 8.1, typically includes only HWuK-related system calls, if needed, it could also include other SWuK system calls and regular software code.

In order to reduce time overheads, hardware tasks are assigned the highest priority in SWuK and thus, they are immediately executed by SWuK’s scheduler when they are ready. Furthermore, the scheduling policy of FreeRTOS is modified in order to disable preemption of hardware tasks. Immediately after a hardware task is inserted in the ready queue of the Task BRAM in HWuK (insert_task() system call), it is blocked in the software level (wait_scheduling() system call) until either the HWuK’s scheduler selects it to be executed on the FPGA or it misses its deadline. When the hardware task is scheduled by HWuK’s scheduler, and once the allocator has found an allocation for it, the task is awakened in the software level. After transferring the input data to be processed by the task to the ODB (wr_ODB() system call), the task is again blocked in the software level (wait_com-
puting() system call, but starts computing in hardware. When the latter computation finishes, the software task is awakened and retrieves the computed results from the IDB (rd_IDB() system call). It is important to note that the aforementioned task blocking and awakening mechanisms in the software level are based on binary semaphores provided by FreeRTOS. The semaphores are taken when executing wait_scheduling() or wait_computing() system calls, and they are released upon reception of the suitable control commands from HWuK. Hence, two arrays of semaphores are kept in SWuK, whose sizes are equal to the number of hardware tasks defined in the application. In these arrays, each semaphore is associated to a specific hardware task by means of a corresponding TaskID. The interaction between software and hardware levels during the life cycle of a task is shown in Figure 8.4.

Listing 8.1: Representation of a hardware task in the SWuK

```c
void vTask_ID (void *pvParameters)
{
    insert_task(task_ID, task_params);
    wait_scheduling(task_ID);
    wr_ODB(&input_data_base_addr, length);
    wait_computing(task_ID);
    rd_IDB(&output_results_base_addr, length);
}
```

Figure 8.4: Management of a hardware task in SWuK

### 8.2.2 High-Level Services offered by SWuK API

The services offered by SWuK are basically those implemented by FreeRTOS, which are extended with support for accessing the low-level HWuK services explained in chapter 7. Table 8.1 shows the most important system calls available in the SWuK API. We note that there is still some work to be done in order to improve the integration of hardware and software worlds. Namely, the system calls to manage hardware tasks should be integrated into the standard SWuK system calls. For instance, wr_ODB() and rd_IDB() functions should be called directly from the message queue management routines of FreeRTOS.
<table>
<thead>
<tr>
<th>Service</th>
<th>System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MANAGEMENT OF SOFTWARE TASKS (FreeRTOS):</strong></td>
<td></td>
</tr>
<tr>
<td>Create a task</td>
<td>xTaskCreate()</td>
</tr>
<tr>
<td>Delete a task</td>
<td>xTaskDelete()</td>
</tr>
<tr>
<td>Stop a task</td>
<td>vTaskSuspend()</td>
</tr>
<tr>
<td>Resume a task</td>
<td>vTaskResume(), xTaskResumeFromISR()</td>
</tr>
<tr>
<td>Enter critical section</td>
<td>taskENTER_CRITICAL()</td>
</tr>
<tr>
<td>Exit critical section</td>
<td>taskEXIT_CRITICAL()</td>
</tr>
<tr>
<td>Force a context switch</td>
<td>taskYIELD()</td>
</tr>
<tr>
<td>Create a queue</td>
<td>xQueueCreate()</td>
</tr>
<tr>
<td>Delete a queue</td>
<td>xQueueDelete()</td>
</tr>
<tr>
<td>Post an item to a queue</td>
<td>xQueueSend(), xQueueSendFromISR()</td>
</tr>
<tr>
<td>Receive an item to a queue</td>
<td>xQueueReceive()</td>
</tr>
<tr>
<td>Create a semaphore</td>
<td>vSemaphoreCreateBinary()</td>
</tr>
<tr>
<td>Delete a semaphore</td>
<td>vSemaphoreDelete()</td>
</tr>
<tr>
<td>Take a semaphore</td>
<td>xSemaphoreTake(), xSemaphoreTakeRecursive()</td>
</tr>
<tr>
<td>Release a semaphore</td>
<td>xSemaphoreGive(), xSemaphoreGiveFromISR(), xSemaphoreGiveRecursive()</td>
</tr>
<tr>
<td><strong>MANAGEMENT OF HARDWARE TASKS:</strong></td>
<td></td>
</tr>
<tr>
<td>Insert a task in Ready queue in HWuK</td>
<td>insert_task()</td>
</tr>
<tr>
<td>Block a task until it is scheduled in HWuK</td>
<td>wait_scheduling()</td>
</tr>
<tr>
<td>Copy data to ODB</td>
<td>wr_ODB()</td>
</tr>
<tr>
<td>Block a task until it completes the computation</td>
<td>wait_computing()</td>
</tr>
<tr>
<td>Read data from IDB</td>
<td>rd_IDB()</td>
</tr>
</tbody>
</table>

**Table 8.1:** Consumed FPGA resources by each R3TOS component
8.3 Chapter Conclusion

This chapter covered the way R3TOS HWuK is wrapped with a SWuK to ease the development of reconfigurable applications by providing an easy and intuitive access to the underlying FPGA resources. The SWuK is based on one of the currently most popular commercially available real-time microkernels (FreeRTOS) which is minimally modified to dispatch hardware tasks to HWuK and to support data transfers between software and hardware levels. Despite the fact that the integration between the HWuK and SWuK is currently loose, it still serves to demonstrate the feasibility of programming reconfigurable applications in a high-level framework using R3TOS as the execution environment.

The chapter also described the main CPU, in which SWuK runs. Currently this CPU is based on a Xilinx MicroBlaze soft-core, but more powerful standard processors are envisaged in future releases of R3TOS on state of the art FPGA platforms (e.g., an ARM dual-core Cortex in a Xilinx Zynq FPGA).
Chapter 9

R3TOS Prototyping and Application Development

After having described the major capabilities of R3TOS in previous chapters, this chapter is aimed at illustrating the applicability of our solution in two real-world scenarios, which are drawn from the Software Design Radio (SDR) and railway transport fields. First, the chapter describes the implemented proof-of-concept R3TOS prototype in terms of both consumed resources and measured performance results.

9.1 Proof-of-Concept R3TOS Prototype

The R3TOS prototype has been implemented on a Xilinx Virtex-4 XC4VLX160 FPGA, which is the device targeted in the simulation environment described in chapter 5 to evaluate the scheduling and allocation algorithms presented in this thesis. This device includes up to 12 clock regions, 88 CLB columns, 7 BRAM columns and 1 DSP48 column. The layout of this chip is very interesting as it includes a 28 CLB column wide homogeneous sandbox in the central part of the chip with the heterogeneous resources located in the edges; in the leftmost edge there are 3 BRAM columns and 1 DSP48 column, while in the rightmost edge there are 4 BRAM columns. Note that edges are very similar in layout to other smaller Virtex-4 devices, where the BRAM and DSP48 columns are sandwiched in-between several

Part of this chapter has been accepted for publication in [Iturbe et al., 2013b].
CLB columns. Besides, the central sandbox allows for arbitrary reallocation of exclusively CLB-based tasks. In order to ease R3TOS prototyping, the commercial board manufactured by Xilinx Inc. depicted in Figure 9.1 is used.

As shown in Figure 9.2, the R3TOS core circuitry is located in the upper-right quadrant of the chip, leaving 3/4 of the FPGA free to allocate the hardware tasks, i.e., Partially Reconfigurable Regions (PRRs). Note that only the clock distribution lines span across the PRRs to reach the regional clocking resources (i.e., BUFRs), which are located in the leftmost and rightmost IOB columns. Next to the BUFRs and, occupying only one CLB column, are the associated diagnostic circuits. The R3TOS circuitry and the clocking resources together make up the R3TOS static infrastructure which is described in detail in chapter 6 (see Figure 6.3).

For simplicity and clarity purposes, the implementation shown in Figure 9.2 does not consider board-dependent IOB location issues. However, in the real implementation using the Virtex-4 LX development kit, the pins to access the external memory are located in the central part of the FPGA die and therefore, other clock regions are occupied with static routes. In any case, the left part of the FPGA (PRR_A) is completely free of static routes.

The R3TOS core circuitry comprises two different parts: (1) the HWuK, which spans 2 FPGA rows in height, and (2) the main CPU (i.e., MicroBlaze), which spans 4 FPGA rows. Both R3TOS parts communicate with each other as well as with I/O device pins through a set of Bus Macros (BMs) located in the rightmost 4 CLB columns of the chip. Indeed, we note that all of the used IOBs (i.e., 69) could be mapped to the upper-right quadrant. This modular implementation of R3TOS favours adaptability and upgradability, as the main CPU
Figure 9.2: FPGA implementation of R3TOS proof-of-concept prototype
(Standard PicoBlazes are used in HWuK and IOB location limitations are not considered)
can be replaced without having to modify the HWuK. In addition, the HWuK includes a set of BMs in the leftmost side to access the FPGA’s hard primitives which are located in the central part of the chip (i.e., Frame_ECC, ICAP, STARTUP and BUFGCTRLs).

### 9.1.1 Implementation Results and Consumed FPGA Resources

Table 9.1 shows the amount of FPGA resources required by each R3TOS component, where the numbers in brackets indicate the resources consumed when using our FT-PicoBlaze instead of regular Xilinx PicoBlaze (see Appendix A). Both main R3TOS parts (i.e., HWuK and main CPU) consume a similar amount of resources, with the configuration manager being the component that requires more logic in comparison with other components. We note that the design of this component is not optimised and hence the margin for improvement is still great.

### 9.1.2 Performance Results

Table 9.2 shows the most significant performance figures in the R3TOS prototype when invoking the system calls from both the Hardware Abstraction Layer (HAL) and Application Programming Interface (API). These results were obtained when clocking the prototype at 100 MHz, with the HWuK tick ($t_{K,T}$) equal to 100 µs and the SWuK tick equal to 1 ms. It can be seen the achievable efficiency improvement when directly using the HAL instead of API. In the worst-case, up to 183 µs are required due to API’s own operations. This overhead includes: (1) the time needed to perform data transfers to/from data buffers, (2) the delay introduced by FreeRTOS when processing interruptions, and (3) communications between main CPU and HWuK.

Updating task queues and Empty Area Descriptor (EAD) are the two processes which require the longest amount of time to be completed in R3TOS. The other R3TOS processes take significantly shorter time to be executed, excepting the time consuming diagnostic test which is used to detect damaged resources and thus is less critical as it is rarely executed. Notably, the amount of time needed by the scheduler and allocator to update the task queues and EAD is slightly shorter (i.e., hundreds of microseconds) of that needed to set-up a typical hardware task using the ICAP (usually several hundreds of microseconds or few milliseconds [Liu et al., 2009]), making it possible to reduce the time overheads in-
### Table 9.1: Consumed FPGA resources by each R3TOS component

<table>
<thead>
<tr>
<th>Component</th>
<th>Slices</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SCHEDULER</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduler PicoBlaze</td>
<td>118</td>
<td>1</td>
</tr>
<tr>
<td>Task BRAM</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>Kernel Timer</td>
<td>36</td>
<td>-</td>
</tr>
<tr>
<td><strong>ALLOCATOR</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocator PicoBlaze</td>
<td>112</td>
<td>1</td>
</tr>
<tr>
<td>FPGA State BRAM</td>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>Architecture Checker</td>
<td>46</td>
<td>-</td>
</tr>
<tr>
<td>Empty Area Descriptor Updater</td>
<td>284</td>
<td>-</td>
</tr>
<tr>
<td>Allocation Quality Evaluator</td>
<td>33</td>
<td>-</td>
</tr>
<tr>
<td><strong>CONFIGURATION MANAGER</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration manager PicoBlaze</td>
<td>108</td>
<td>1</td>
</tr>
<tr>
<td>ICAP Driver</td>
<td>507</td>
<td>-</td>
</tr>
<tr>
<td>Bitstream BRAM</td>
<td>44</td>
<td>1</td>
</tr>
<tr>
<td>Configuration Guardian</td>
<td>38</td>
<td>-</td>
</tr>
<tr>
<td>CRC32 Module</td>
<td>125</td>
<td>-</td>
</tr>
<tr>
<td>STARTUP Logic</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td><strong>DIAGNOSIS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock Diagnostic</td>
<td>144</td>
<td>-</td>
</tr>
<tr>
<td>Communication Monitors</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td><strong>MISCELLANEOUS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>External Memory Ctrl.</td>
<td>104</td>
<td>-</td>
</tr>
<tr>
<td>Interruption Manager</td>
<td>23</td>
<td>-</td>
</tr>
<tr>
<td>Bus Macros</td>
<td>94</td>
<td>-</td>
</tr>
<tr>
<td><strong>TOTAL R3TOS HWuK</strong></td>
<td>2,003</td>
<td>6</td>
</tr>
<tr>
<td><strong>TOTAL MAIN CPU</strong></td>
<td>2,392</td>
<td>24</td>
</tr>
<tr>
<td><strong>TOTAL R3TOS CIRCUITRY</strong></td>
<td>4,401</td>
<td>30</td>
</tr>
</tbody>
</table>

1 The data buffers are: 1 CLB column (2 Kb) and 1 BRAM column (64 Kb).

279
## Table 9.2: Performance figures in the R3TOS prototype

<table>
<thead>
<tr>
<th></th>
<th>HAL</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min.</td>
<td>Max.</td>
</tr>
<tr>
<td><strong>TASK MANAGEMENT</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task execution overhead</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Read/Write from/to data</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>SCHEDULING</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling algorithm</td>
<td>&lt; 1 µs</td>
<td>&lt; 100 µs</td>
</tr>
<tr>
<td>Queues and task state</td>
<td>&lt; 1 µs</td>
<td>&lt; 300 µs</td>
</tr>
<tr>
<td><strong>ALLOCATION</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocation algorithm</td>
<td>&lt; 1 µs</td>
<td>&lt; 100 µs</td>
</tr>
<tr>
<td>Empty Area Descriptor</td>
<td>10 µs</td>
<td>&lt; 200 µs</td>
</tr>
<tr>
<td><strong>DEALLOCATION</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deallocation of a</td>
<td>2.3 µs</td>
<td>33.2 µs</td>
</tr>
<tr>
<td>hardware task</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TASK PREEMPTION</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disable writing to all</td>
<td>&lt; 2.5 ms</td>
<td>&lt; 2.5 ms</td>
</tr>
<tr>
<td>flip-flops at start-up</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context save / restore</td>
<td>26.6 µs</td>
<td>26.6 µs</td>
</tr>
<tr>
<td>**INTER-TASK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMUNICATIONS**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer LUT data buffer</td>
<td>3.7 µs</td>
<td>3.7 µs</td>
</tr>
<tr>
<td>(ICAP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer BRAM data buffer</td>
<td>60.18 µs</td>
<td>60.18 µs</td>
</tr>
<tr>
<td>(ICAP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer BRAM data buffer</td>
<td>36.18 µs</td>
<td>39.9 µs</td>
</tr>
<tr>
<td>(DRTs): Conf.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer BRAM data buffer</td>
<td>81.92 µs</td>
<td>81.92 µs</td>
</tr>
<tr>
<td>(DRTs): Func.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switching BRAMs between</td>
<td>10.03 µs</td>
<td>10.03 µs</td>
</tr>
<tr>
<td>neighbour tasks</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INTER-TASK SYNCHRONISATION</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polling of a HWS</td>
<td>1.6 µs</td>
<td>1.6 µs</td>
</tr>
<tr>
<td>Activation of a HWS</td>
<td>3.7 µs</td>
<td>3.7 µs</td>
</tr>
<tr>
<td><strong>CLOCKING MANAGEMENT</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable/Disable a BUFR</td>
<td>≈ 4 µs</td>
<td>≈ 4 µs</td>
</tr>
<tr>
<td>Adjust task clock</td>
<td>≈ 4 µs</td>
<td>≈ 4 µs</td>
</tr>
<tr>
<td>frequency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switch a spare clock</td>
<td>≈ 4 µs</td>
<td>≈ 4 µs</td>
</tr>
<tr>
<td>source in a BUFR</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RELIABILITY</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute the diagnostic</td>
<td>167.5 µs</td>
<td>4.9 ms</td>
</tr>
<tr>
<td>test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scrub a configuration</td>
<td>4.81 µs</td>
<td>9.34 µs</td>
</tr>
<tr>
<td>frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate a Frame_ECC</td>
<td>47.2 µs</td>
<td>47.2 µs</td>
</tr>
</tbody>
</table>

1. It refers to the time needed to execute a task with \( t_{ICAP,i} = t_{E,i} = 0 \). The time until it is scheduled in HWuK is not considered.
2. The data buffers are considered of minimal size: 1 CLB column (2 Kb) and 1 BRAM column (64 Kb).
3. A maximum of 10 ready and executing tasks were considered.
4. All of the 60 tasks change their state.
5. The size of the task to deallocate ranges from a single CLB column to a complete sandbox occupying task.
6. The task to preempt spans only 1 CLB column. It is assumed there is access to external memory.
7. The size of the Region Under Test (RUT) ranges from a single CLB column to the complete sandbox.
roduced by R3TOS as these three processes can be concurrently carried out in most of the cases. The obtained results are promising in light of enabling the use of our solution with newer reconfigurable technology with presumably faster reconfiguration capabilities and larger sizes.

We acknowledge the existence of a time overhead which is introduced by R3TOS when making the scheduling and allocation decisions. Since these decisions are mostly made by software routines in the allocator and scheduler PicoBlazes, the overhead can reach up to tens of microseconds per each task allocation attempt. Note that although the allocator PicoBlaze relies on the Allocation Quality Evaluator (AQE) to accelerate the allocation process, it is still responsible for exploring the EAD to find feasible allocation candidates to be evaluated, thus limiting the achievable acceleration. In Table 9.2, the scheduling decisions making time is given when there are 10 tasks in the ready and executing queues, and the queues updating time is given in the critical instant when all of the 60 tasks in the Task BRAM change their state. As for the allocator, it would be convenient to use a hardware accelerator in the scheduler when dealing with a larger number of tasks in order to keep the task management overheads within reasonable bounds.

Table 9.2 also shows the achievable acceleration when exchanging data among tasks using DRTs or directly accessing the data in producer task's ODB. While the time needed to transfer the content of a BRAM-based data buffer using the ICAP is about 60 microseconds, the access to the BRAM can be switched between two tasks within only 10.03 microseconds (around 6x speed-up). In addition, 36.18 microseconds are needed to configure a DRT (around 1.6x speed-up), which then requires 81.92 microseconds to complete the data transfer through the functional layer. Note that the latter time does not constrain the performance as it can be parallelised with the task configuration phase.

A hardware accelerator to speed-up the Frame_ECC computation is an aspect to discuss in the future. These computations are currently performed by the configuration manager's PicoBlaze, delaying a significant time a task relocation. Namely, while only 3.7 $\mu$s are needed to relocate a configuration frame, up to 47.2 $\mu$s are required to compute its Frame_ECC. In any case, as occurs with the diagnostic test, this is not critical as Frame_ECC computation is rarely required, only when a task needs to be allocated in-between clock regions.
9.2 Control of the Power Chain of a Railway Traction Vehicle

This section explains the potential of R3TOS in high-reliability applications. To demonstrate this, we have developed a R3TOS-based simplified traction controller to drive a motor in a railway locomotive. The reason to choose this application is the strong background that IK4-Ikerlan has in developing railway solutions, mainly for the Basque train manufacturer CAF SA\(^1\). However, we note that the reliability offered by R3TOS is more oriented to applications that require long life time (longevity) rather than to those which require availability, as it is the case of railway equipment.

9.2.1 Application Description

A motor is considered a device that transforms electrical energy into mechanical motion. The current that is applied to the motor flows through a winding with wires, creating an electromagnetic field that exerts a torque force on the axis of the motor. The more current that is applied to the traction motor of the locomotive, the more force the latter will apply on the wheels. On the other hand, when low or no current is applied, the inner resistance of the motor acts as a brake on the locomotive. While this continues to be valid for a 3-phase AC motor, i.e., the force exerted by the motor is proportional to the Root Mean Square (RMS) supply current, the speed at which it turns is determined by the frequency of its supply. In modern railway applications, AC motors are preferred over DC ones because they are cheaper, more robust, and easier to maintain.

The amount and frequency of current that is applied to an AC traction motor is typically controlled with a Pulse Width Modulated (PWM) signal that feeds a 3-phase IGBT-based rectifier bridge\(^2\). The latter receives its power from the power line catenary through train’s pantograph. Hence, our simplified model of the power chain of the railway vehicle starts in the pantograph and finishes in the traction motor.

As shown in Figure 9.3a, the 6 IGBTs of the rectifier bridge are organised in three arms, where the two IGBTs in each arm switch on and off many times to produce each phase required by the AC motor. The on/off switching of IGBTs is restricted by a set of rules. For

---

\(^1\)CAF: Construcciones y Auxiliar de Ferrocarriles (http://www.caf.es/ingles/home).

\(^2\)An Insulated Gate Bipolar Transistor (IGBT) is a switching transistor that requires little voltage on the gate to conduct strong currents.
instance, the two IGBTs in an arm can never be enabled at the same time, otherwise a short-
cut circuit occurs. In order to ensure meeting this, a dead time is allowed during which both
IGBTs in the arm remain disabled prior to changing the one that is conducting. Moreover,
the IGBTs must be enabled for a minimum amount of time in order to prevent them from
overheating.

Based on both the traction/braking orders given by the driver and the measured speed
of the train at any time, a traction controller must tweak the width of the generated PWM
pulses, under a closed-loop control, to derive to the motor the specific amount of power and
at the proper frequency to produce the required motion with precision. In the current so-
lution developed by IK4-Ikerlan and CAF SA consortium, this functionality is implemented
using a combination of a Digital Signal Processor (DSP), which executes the traction control
algorithm, and an FPGA, which creates the corresponding PWM pulses and implements the
IGBT protection (e.g., dead times). This is done with a time resolution as high as 500 ns.

Our aim has been to demonstrate the feasibility of implementing an equivalent R3TOS-
based traction controller using exclusively an FPGA. Notably, our traction controller is able
to continue operating correctly when faults are injected into the configuration memory of
the FPGA on which it is implemented. In order to do so, it relocates the circuitry on-the-
fly, circumventing the damaged resources on the chip. Unlike similar previous approaches
(e.g., [Montminy et al., 2007]), our solution is successful because it does not rely on any
physical routes to connect the relocatable circuitry and the output FPGA pins; it uses FPGA's
ICAP for that purpose.

We acknowledge that the currently implemented traction control algorithm is very basic
and hence not suitable to be used in a real train as-is. To enable its use in a real scenario,
the control algorithm should account for several key aspects (currently neglected), such as
acceleration compensation. However, we posit that considering such aspects do not involve
any extra difficulties beyond those found in standard digital design and therefore, the lim-
ited control algorithm currently used still serves to demonstrate the potential of R3TOS to
develop real-world industrial applications with high-reliability requirements. We also ac-
knowledge that, in order to enable the use of our solution in a safety-critical application
such as railway transport, TMR should be used to tolerate common source failures in the
FPGA chips (e.g., power supply or clock source failures). Hence, at least three traction con-
trollers like the one herein described should be included in a train. Moreover, the redundant controllers should be preferably located in physically separated zones to reduce the risk of collective damage due to mechanical forces.

9.2.2 Application Development

The developed traction controller is based on the R3TOS implementation described in section 9.1, which is completed with a minimal static circuitry aimed at handling the system’s inputs/outputs (see Figure 9.3a). The input circuit is responsible for receiving a target train speed that is computed by the cabin controller subsystem based on the orders given by the driver and the odometry-related information (e.g., state of the railway and recommended speed in each stretch). Besides, the input circuit also receives the train speed that is measured by an external sensor, i.e., Wheel Impulse Generator (WIG). On its part, the output circuit is in charge of issuing the PWM signals that control the power delivered to the traction motor through FPGA pins. Each of these pins drives the gate of one IGBT in the rectifier bridge.

Both the input and output circuits are based on LUTs, which are used as SRL16. The 16 bits indicating the train speed delivered by the speed sensor are serially multiplexed to fit in an input SRL16 as shown in Figure 9.3b. A similar scheme is to be used to capture the target speed value delivered by the cabin controller subsystem, which is indeed coded using 16 bits as well. Likewise, the LUTs included in the output circuit are directly connected to the PWM pins (see Figure 9.3c), thus minimising the probability of suffering an error. The reason to use SRL16 in the output circuit are twofold. First, performance: PWM samples are transferred in units of 16 bits to allow for a sufficiently high throughput to achieve a time resolution of 500 ns. As shown in Figure 9.3d, two SRL16s that operate in a ping-pong fashion, odd and even, are assigned to each PWM pin. While one of the SRL16s delivers its 16 PWM bits to the FPGA pin, the other SRL16 is loaded through the ICAP with the next 16 PWM bits to be transmitted. Note that the latter functioning, which permits to hide the lateness of ICAP-based communications, requires that both SRL16s assigned to the same pin are mapped to different CLB frames in order to avoid corruption when accessing them through the ICAP. Second, reliability: the serial data input port of the SRL16s is connected to '0' and thus, in case R3TOS gets stuck and stops updating the PWM bits in the SRL16s, the
Figure 9.3: R3TOS-based traction controller
latter will deliver '0's after finishing outputting the lastly updated 16 PWM bits. Note that a permanent zero PWM output prevents the IGBTs from conducting, making locomotive's traction motors stop and thus guaranteeing the safety of the train and its passengers.

The block diagram of the relocatable circuit that performs the traction control algorithm and generates the PWM signals for the IGBTs is shown in Figure 9.4a. We name this circuit as PC-PWM, where PC stands for Proportional Control as it implements a proportional closed-loop control, where the measured train speed acts as the feedback. Hence, the measured speed is subtracted from the target speed value to produce an error signal at every control loop, $T_{LOOP}$. Note that both the measured speed and the target speed values are delivered to the circuit via input LUTs, which need to be de-serialised as the input circuit fits the speed values into SRL16s. The aforementioned error signal is used to calculate the next output power and frequency to be applied to the traction motor, which finally modulates the corresponding PWM pulses. In order to do so, the circuit includes a Numerically Controlled Oscillator (NCO) that is dynamically adjusted to produce three reference sinusoidals of the required amplitude and frequency (i.e., each sinusoidal is shifted 120° in phase from the others). As shown in Figure 9.4b, the reference sinusoidals are compared with a triangular signal to generate the PWM pulses, i.e., the PWM signals change their state every time the triangular crosses the sinusoidals in such a way that they fulfil the on/off switching rules of the IGBTs. In this context, note that the PWM switching period, $T_S$, matches the period of the used triangular signal, while the PWM time resolution is determined by the period of the used clock (i.e., $T_{PC-PWM-CLK} = 500$ ns). The currently implemented PC-PWM circuit works with 208 duty cycle increments and therefore, the PWM switching frequency is 208 times slower than the used clock frequency (i.e., $f_S = f_{PC-PWM-CLK}/208 9.6$ KHz). Moreover, the traction control loop is closed every 10 PWM switching periods (i.e., $T_{LOOP} = 10 \cdot T_S \approx 1$ ms).

The value of the PWM signals at every clock cycle are written to the output LUTs in the PC-PWM circuit, which implement its output interface. The latter LUTs mimic the scheme used in the traction controller’s output circuit, i.e., two SRL16s working in a ping-pong fashion. Hence, the 16 PWM bits stored in one of the LUTs are accessed through the ICAP to be transferred to the output circuit, while the other LUT is filled with the new 16 PWM bits to be next transmitted. Again, this functioning requires that both LUTs associated to the same
Chapter 9- R3TOS Prototyping and Application Development

(a) Block diagram

(b) Functioning (Only one phase is depicted and PWM dead times are not shown)

(c) FPGA implementation

Figure 9.4: PC-PWM circuit
PWM signal are mapped to different CLB frames. As the PC-PWM works with 208 duty cycle increments, the 208 PWM samples corresponding to one switching period are transferred in 13 units of 16 bits through the ICAP.

The PC-PWM circuit also includes `error_flag` LUTs to indicate any deviation from its expected functioning. This includes violations of the on/off switching rules detected in the generated PWM signals (e.g., simultaneous enabling of complementary pairs of IGBTs, no pulse generation within a PWM switching period, or pulse duration shorter than minimum pulse width) and errors diagnosed by the circuit itself (e.g., double errors in BRAM data words or illegal states in FSMs). In order to ease the access to the `error_flag`, these LUTs are mapped to the same columns as the aforementioned circuit’s output PWM LUTs.

With the objective of increasing the availability of the system and diagnosing functioning errors, three instances of the PC-PWM circuit are kept running on the FPGA (see Figure 9.3a); i.e., TMR is used. For simplicity, the FPGA is not divided into Fault Containment Computation Regions (FCCRs).

Note that the fact that the triplicated PC-PWM circuit instances work synchronised is vital to ensure the efficacy of the adopted TMR scheme. Synchronisation is required after recovering any instance of the circuits from an error, i.e., when relocating the circuitry to another position on the FPGA. Unlike in standard R3TOS implementation, where the synchronisation mechanism is based on a LUT (i.e., HWS) which is checked when the tasks finish executing, the implemented mechanism in this application, where the PC-PWM circuit can never be stopped, is based on flip-flops. Using FPGA’s Global Set Reset (GSR) internal signal, it is indeed possible to force a reset in selected flip-flops of the three PC-PWM circuits at the same time (see chapter 6). The flip-flops used to implement the synchronisation mechanism (SYNC) are connected in a chain as shown in Figure 9.4a. They are configured with `SRMODE='0'` (i.e., their output is ‘1’ when GSR is toggled), and the input of the first flip-flop in the chain is connected to ‘0’. Thus, in the next clock cycle after having toggled the GSR signal, the latter flip-flop is loaded with ‘0’ and two clock cycles after, that ‘0’ propagates to the second flip-flop in the chain. The reset is generated by ORing the output of both flip-flops, ensuring that it remains in high for at least one clock cycle. Note that the only frames that must be sensitive to the GSR signal (i.e., mask bit equal to ‘0’) are those where the SYNC logic is mapped to, and the use of the remaining flip-flops in those frames must be prohibited.
The proper instant to synchronise the triplicated PC-PWM instances is when the reference sinusoidal with phase 0° crosses through 0. Note that this instant must be predicted to compensate for the (fixed and known) delay introduced by R3TOS when toggling the GSR signal. In order to do so, a Zero Cross Predictor logic is included in the PC-PWM circuit, which writes its output (\texttt{Sync\_flag}) to a couple of LUTs mapped to the same frames as the PWM LUTs. Hence, \texttt{Sync\_flag}='1' indicates the proper instant to start the execution of the routine to toggle the GSR signal. Note that, the fact that the value of the three \texttt{Sync\_flags} related to the PC-PWM instances do not match means that the system has lost the synchronisation.

Figure 9.4c shows the baseline implementation of the PC-PWM circuit, spanning the bottom half of an FPGA row and being clocked from a BUFR. Note that the input and output LUTs, as well as the SYNC logic, are allocated to dedicated columns in the rightmost and leftmost edges of the circuit. It is significant that the PC-PWM circuit uses 1 BRAM, to implement the Look-Up-Table (LUT) where the normalised samples corresponding to one period of the reference sinusoidals are stored, and 3 DSP48s, to adjust the amplitude of the latter sinusoidal signals (see Figure 9.4a). As a result, the circuit can only be relocated vertically along the leftmost edge of the FPGA, namely to 48 possible target locations (i.e., 4 locations inside each clock region, and 12 clock regions on the FPGA). Without the capability to allocate a partial bitstream in-between clock-regions, the PC-PWM circuit could only be relocated to a maximum of 12 positions.

9.2.3 Application Functioning

As the three PC-PWM circuit instances are permanently executing, it makes no sense to refer to them as hardware tasks. Consequently, the developed traction controller does not use the R3TOS scheduling functionality and thus, the Task BRAM is used exclusively to keep track of the location of the PC-PWM instances at every time. On the other hand, R3TOS allocator plays a key role in this application to find a fault-free location where to relocate any PC-PWM circuit instance when it is affected by a permanent fault.

In this application, R3TOS is responsible for carrying out ICAP-based communications between the PC-PWM circuit instances and the input/output LUTs, as well as for ensuring the correctness of the system functioning. As shown in Figure 9.6, the latter function-
ality is implemented in cyclic ICAP phases of duration $T_{ICAP}$. Indeed, it is of outmost importance to ensure that ICAP phases are correctly synchronised with the functioning of PC-PWM instances, i.e., each ICAP phase must span exactly 16 PC-PWM clock cycles ($T_{ICAP} = 16 \cdot T_{PC-PWM-CLK} = 8 \, \mu s$).

The synchronisation between the ICAP phases and the PC-PWM instances is accomplished by means of interruptions. More specifically, a 4-bit counter that works with the same clock signal used in the PC-PWM instances and that is connected to the same reset circuit as used in the PC-PWM instances (i.e., flip-flop chain sensitive to the GSR signal, see Figure 9.4a) generates an interruption to the R3TOS configuration manager at the beginning of every ICAP phase.

The developed traction controller can be thus compared with a time-triggered system, where the micro-tick period is equal to $T_{PC-PWM-CLK}$ and the macro-tick period is equal to $T_{ICAP}$. Figure 9.5 outlines the relation among the principal periods of time in the developed traction controller.

As shown in Figure 9.6, every ICAP phase allocates time for reading-back the PWM bits from the output LUTs of the PC-PWM circuit instances and for writing the majority voted values to system’s output LUTs (i.e., PWM sample chunks). Additionally, each ICAP phase allocates a time slot to carry out other custom operations which can be executed at a slower rate. Namely, these operations are: (1) deliver input information to the PC-PWM circuit instances, i.e., target speed and measured feedback speed, (2) check R3TOS configuration correctness using Frame_ECCs, (3) relocate erroneous PC-PWM circuit instances to fault-free locations on the FPGA, or (4) inject faults to the system.
Figure 9.6: ICAP phase
Figure 9.7: Ping-pong functioning of PWM sample chunks
The input information is delivered to the PC-PWM circuit instances at every traction control loop, $T_{LOOP}$. As the information is the same for the three circuit instances, the WMF2S ICAP function is used with the objective of speeding-up communications (see chapter 6). As shown in Figure 9.6, the two ICAP transfers required to deliver the input information (i.e., an RBF followed by a WMF2S), give raise to a type A custom operation. As the traction loop is closed every 10 PWM switching periods and, as each period defines 208 duty cycle increments (i.e., 13 ICAP transfers), type A operations are executed every 130 ICAP phases (i.e., $T_{LOOP} = 10 \cdot T_S = 130 \cdot T_{ICAP}$).

Type B operations are mainly used to check the correctness of Frame_ECCs (i.e., RBF followed with computing time). In the case that a soft-error needs to be corrected, the subsequent writing leads to a Type C custom operation. The latter consist of WSF2S followed by RBF and is the most generic operation as it is used either when scrubbing, when relocating a PC-PWM circuit or when injecting faults.

Type C operations also include a computing time, which is used to invert a bit in the read-back frame when injecting a fault, to modify some specific configuration information (e.g., configure a BUFR) when relocating a PC-PWM instance, or to check the correctness of the Frame_ECC associated to the read-back frame.

However, the amount of time needed to correct an erroneous configuration frame based on the Frame_ECC syndrome, or to compute a new Frame_ECC value for a relocated frame, usually exceeds the duration of the ICAP phases. Therefore, these computations are scheduled in the next ICAP phases, leading to Type D custom operations. Thus, Type D operations are always alternated with Type B and C, to which they complement.

Finally, Type E custom operations are used to synchronise the three PC-PWM circuit instances by toggling the GSR signal, and they are executed only when a synchronisation instant is detected.

Due to the strict timing requirements when synchronising the three PC-PWM circuits, Type E operations are assigned the highest priority. The second highest priority is for Type A operations. For instance, the relocation of a PC-PWM circuit instance is momentarily

---

3WMF2S: Write Multiple Frames to Single location.
4RBF: Read-Back a frame.
5WSF2S: Write Single Frame to Single location.
interrupted to deliver the input speed values to the other running circuit instances. Type B, C and D operations are executed in the remaining time as needed.

In order to reduce the overheads, the traction controller directly invokes R3TOS system calls from the HAL; i.e., the R3TOS main CPU is not used. Furthermore, the configuration templates of the next ICAP operations to be executed are adjusted (e.g., write FAR address and number of words to be read-back/written) while the previous configuration operations are being performed. This allows to complete the ICAP phases within 8 $\mu$s, as required in this application (see Figure 9.4b). In addition, we note that this is fast enough to satisfy the timing requirements imposed by most of control algorithms currently used in the industry.

**Reliability**

The reliability of the developed traction controller is supported by four different mechanisms that complement each other: (1) TMR, (2) relocation, (3) scrubbing and (4) diagnosis.

During the voting process, the PWM bits generated by any PC-PWM circuit instance with $\text{error\_flag}=1'$ are not considered. If all of the PC-PWM circuit instances are erroneous, i.e., their $\text{error\_flags}$ are equal to '1' or they have generated different PWM bits, then the system is not trustable and R3TOS writes all zeros to the output LUTs in order to guarantee the safety of the train.

When relocating an erroneous PC-PWM circuit instance to a fault-free location, the configuration information is directly read-back from any of the other circuit instances with $\text{error\_flag}=0'$. As a result, the need for an external memory to store the partial bitstream is circumvented. Indeed, the latter memory is replaced by the FPGA’s own configuration memory, which is protected by means of Frame_ECCs.

Our traction controller is able to localise and correct most of the soft-errors affecting the R3TOS HWuK in less than 10 ms and, it is able to relocate a PC-PWM circuit instance to another position within the FPGA in less than 4 ms. Relocation is only carried out when a PC-PWM circuit instance does not work correctly in the original location (i.e., $\text{error\_flag}=1'$), and the target location is chosen using First-Fit criteria. As explained in chapter 7, the PC-PWM circuitry can be relocated to either another FPGA row, a different position within the same row (e.g., upper half), or it can span over two adjacent rows in the target location.
Note that the former case is preferred as the latter two involve processing the content of the frames to relocate as well as computing new Frame_ECC values for them.

### 9.2.4 Experimental Evaluation

In order to characterise the fault-tolerance capabilities of the developed traction controller, a set of different fault injection campaigns were carried out. The injected faults to the R3TOS HWuK were assumed to be correctable (i.e., soft-errors), while the 0.5% of the faults injected to the PC-PWM circuitry were considered to be non-correctable hardware faults. To simplify, no diagnostic test was used to detect the simulated damage, the corresponding resources were directly marked as damaged in the FPGA State BRAM.

Figure 9.8 shows the observed failure rate during the fault injection campaigns we conducted using the developed R3TOS-based traction controller and a standard controller protected with “static” TMR. We let both systems run for 1 minute, repeating the experiments up to 25 times for each fault rate. The faults were pseudo-randomly injected at equal intervals of time and only in the used FPGA area in each case, i.e., in the area assigned to the PC-PWM circuit instances of the standard controller and, in the area assigned to the PC-PWM circuit instances as well as to R3TOS HWuK of the R3TOS-based controller. However, no faults were injected in the BRAMs arguing that these components will be protected by ECCs in the final R3TOS implementation (see Appendix A).

The set of the frame addresses where to inject a fault was generated offline and stored in a BRAM (i.e., 512 32-bit frame addresses were stored in a single BRAM). On its part, the position of the bit to be corrupted within the frames linearly and continuously varied from 0 to 1,311. In the case of the R3TOS-based traction controller, two list of frame addresses were generated: one for the HWuK and the other for the PC-PWM circuit instances. Note that the addresses stored in the latter list were relative, i.e., referred to the actual position of the PC-PWM circuit in the device. The different size of both parts, and thus the different probability of being affected by a fault, was also considered when injecting the faults. In fact, the list of the frame addresses for the HWuK was approximately 5 times larger than that for the PC-PWM instances, and at runtime, a single fault was injected to each PC-PWM circuit instance per 5 faults injected to the HWuK.
We considered that a failure had occurred either when: (1) all of the error_flags of the PC-PWM circuit instances were activated, (2) all of the PC-PWM circuit instances generated different PWM bits, or (3) R3TOS HWuK was stuck. For the sake of simplicity, we disabled the full FPGA auto-reconfiguration capability of R3TOS.

While the failure rate on the standard controller gradually grows as the injected fault rate increases, the failure rate on the R3TOS-based controller is only significant when the fault injection rate is higher than 32 faults per second (see Figure 9.8). For injection rates higher than this value, the time needed to completely check (and scrub) the configuration of the R3TOS HWuK approaches the interval of time at which faults are injected and therefore, faults start to accumulate eventually leading to R3TOS HWuK failure. For injection rates lower than 32 faults per second, the system fails only when faults (randomly) affect critical elements in the HWuK. We note that a fault rate as high as 32 upsets per second is very aggressive and highly unlikely in a real application. For instance, the highest amount of upsets measured by NASA's spacecraft operating at low Earth orbit (700 km) in one day is about 1,200 (less than one upset per minute) 6.

### SDR Application

This section explains the way R3TOS helps developing SDRs with dynamically changeable characteristics, such as communication standard, modulation, speed, or transmission spectrum [Mitola, 1993, Mitola, 1995, Jondral, 2005]. SDRs can adapt their functionality de-

---

6The referred set of data was collected on a large number of devices for a 4.3-year period (from January 1, 1999, to April 6, 2003) [Poivey et al., 2003].
pending on the environment with the objective of ensuring safe, uninterrupted and ubiquitous communications. When used together, R3TOS and SDR permit to build highly reliable systems with capability to deal not only with “internal” hazards emerging on chip (i.e., spontaneously occurring faults), but also with “external” threats appearing in the environment where the chip is working (e.g., spontaneous interferences). Furthermore, R3TOS permits to obtain the best performance from the resources on the chip, switching them to perform either communication (i.e., SDR), or computation related tasks at different times. Including some computation capabilities in the SDR transmitter (i.e., data pre-processing), allows for significant energy saving and more reduced communication bandwidth as only the significant pieces of information are to be sent. This is especially useful for applications with size, bandwidth, consumption and resource constraints which operate in remote (and harsh) locations, such as remote sensor networks, deep space exploration spacecraft, or offshore wind turbines.

Another potential niche for the adaptability provided by the R3TOS-SDR association can be found in modern mobile applications, whose networking and computing demands are constantly increasing\(^7\) and straining operating budgets. Modern portable devices are thought to be the answer in a world where people feel the need to be connected, informed, and amused everywhere and at any time. As a result, they must be versatile enough to adapt themselves based on network configuration and availability (e.g., a different set of networks go down and up as the user moves around), and to serve the computing requests triggered by the user (e.g., multimedia or video-gaming).

Some of the capabilities described above have been demonstrated with a R3TOS-based SDR multi-standard transmitter prototype, which is shown in Figure 9.9. The data to be transmitted is digital video which is first compressed using JPEG (i.e., data is pre-processed).

The SDR application described in this section has been mostly developed by Mr Raúl Torrego [Torrego et al., 2012b, Torrego et al., 2012a]. My role in this development consisted only in guiding Raúl in the use of R3TOS. In any case, a description of this application has been included in this thesis in order to better demonstrate R3TOS capabilities and limitations.

\(^7\)A standard cell phone in 2012 is reported to have more computing power than the computers used by NASA during the Apollo era [Kaku, 2011].
9.3.1 Application Description and Development

The prototype is based on the proof-of-concept development presented in section 9.1, which is extended with a superheterodyne Radio Frequency (RF) front-end and a color CIF image sensor\(^8\). As shown in Figure 9.9a, the RF front-end includes A/D and D/A converters, a frequency mixer, band-pass filters and antenna electronics, while the CIF video sensor provides an 8-bit RGB input. Consequently, the R3TOS implementation shown in Figure 9.2 is coupled with some specific static logic to drive the RF front-end (i.e., A/D and D/A controllers), and for receiving the RGB video signal to be transmitted. This logic is allocated below the R3TOS core circuitry, in the bottom-right quadrant of the chip. Central to this logic are ping-pong memories, which are composed of two memories that are alternatively switched between read and write. While one of the memories in the D/A controller is written by R3TOS, the content of the other memory is transmitted through the RF front-end, and similarly, while one of the memories of the A/D controller is read by R3TOS, the other memory is filled with data received from the RF front-end.

Our SDR prototype implements a simplified version of three of the currently most used communication standards: IEEE 802.11 WiFi, IEEE 802.16 WiMAX, and IMT-2000 UMTS (also known as 3G). While UMTS uses Wideband Code Division Multiple Access (WCDMA) modulation, WiFi and WiMAX use Orthogonal Frequency-Division Multiplexing (OFMD) modulation with 64 and 256 sub-carriers, respectively. Unlike in WiMAX, in WiFi some training sequences that are known to the receiver are periodically transmitted in order to estimate the distortion introduced by the communication channel. Notably, our prototype is able to switch the communication standards on-the-fly to achieve smooth transition of the service as the system moves from one network connection to another. However, as we have not implemented the communication standard detection functionality yet, standard switches are forced by pushing some buttons in the prototype.

Additionally, our SDR prototype implements a “home-made” cognitive radio solution which is able to autonomously detect and circumvent any interferences provoked in the transmission channel by using spectral switches. The interferences were generated in selected spectrum bands with an RF white noise source. The implemented modulation in our

\(^8\)CIF (Common Intermediate Format) image sensors have a resolution of 352 x 288 pixels and are typically used in video teleconferencing applications.
Figure 9.9: SDR demonstrator prototype
cognitive radio is Offset Quadrature Phase-Shift Keying (OQPSK). Besides receiving the data to be transmitted, the latter modulator also receives the output Intermediate Frequency (IF), which is computed on-the-fly based on the availability of spectral bands in the transmission channel (it can range between 5 to 10 MHz). Indeed, the Power Spectral Density (PSD) in the transmission channel is periodically evaluated (with a period of 50 ms) using Fourier transform methods to estimate any potential interferences and noise. Refer to [Torrego et al., 2011] for further explanations on this.

In all cases data bit sequences are mapped to a Quadrature Phase Shift Keying (QPSK) constellation. Moreover, Forward Error Correction (FEC) channel encoding (e.g., Reed-Solomon and convolutional) and inter-leaving techniques are used in order to improve the robustness of the communication, and data puncturing technique is used with the objective of achieving a higher transmission speed.

We used the JPEG compressor available under GPL license in http://opencores.org/project,jpeg to pre-process the images captured by the CIF image sensor prior to being transmitted.

Unlike in the traction controller application described in section 9.2, where performance is critical, our SDR prototype is controlled from the R3TOS main CPU in light of further developing it. We posit that the performance loss due to the high abstraction provided by the main CPU is to be compensated with the expected facilities to develop new functionality in a software-like environment.

The SDR functionality was firstly developed and validated using Xilinx System Generator tool. This high-level design tool permits to save time and money in the development as it provides a very useful collection of modules ready to be used. For instance, the mathematical complexity associated with Galois Field arithmetic could be circumvented by using the Reed-Solomon encoder included in the standard communication library of System Generator.

The natural partitioning of System Generator models into (data-stream processing) tasks consists in grouping all of the logic in-between consecutive memories together, using the data buffers included in the R3TOS Task Control Logic (TCL) to implement the latter memories. However, some pieces of the logic are too large and thus need to be decomposed into
a set of smaller tasks. More specifically, the OFDM modulator was divided into three or four smaller tasks: (1) the OFDM symbol builder which allocates the data symbols to orthogonal sub-carriers, (2) the Inverse Fast Fourier Transform (IFFT) to translate the OFDM symbols to the time domain, (3) the Cyclic Prefix Inserter (CPI) to mitigate the inter-symbol interference due to multi-path propagation, and in the case of WiFi, (4) the Training Sequence Inserter (TSI). At this point, it is convenient to clarify that the purpose of this demonstrator is not to come up with the most efficient application partition, but to demonstrate the feasibility of developing R3TOS-based SDR applications. We decided to implement the task-set listed in Table 9.3.

<table>
<thead>
<tr>
<th>ID</th>
<th>Task Name</th>
<th>Slices</th>
<th>BRAMs</th>
<th>DSP48s</th>
</tr>
</thead>
<tbody>
<tr>
<td>θ1</td>
<td>JPEG Compressor</td>
<td>2,859</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>θ2</td>
<td>Randomiser</td>
<td>53</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>θ3</td>
<td>Reed-Solomon Encoder</td>
<td>196</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>θ4</td>
<td>Convolutional Encoder</td>
<td>46</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>θ5</td>
<td>Puncturer</td>
<td>39</td>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>θ6</td>
<td>Inter-leaver</td>
<td>468</td>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>θ7</td>
<td>Data Symbol Mapper</td>
<td>132</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>θ8</td>
<td>WCDMA Modulator</td>
<td>3,041</td>
<td>8</td>
<td>67</td>
</tr>
<tr>
<td>θ9</td>
<td>OFDM 64-carrier Builder</td>
<td>749</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>θ10</td>
<td>Training Sequence Inserter</td>
<td>1,771</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>θ11</td>
<td>64-tap IFFT</td>
<td>1,536</td>
<td>11</td>
<td>26</td>
</tr>
<tr>
<td>θ12</td>
<td>16-word Cyclic Prefix Inserter</td>
<td>59</td>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>θ13</td>
<td>OFDM 256-carrier Builder</td>
<td>3,332</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>θ14</td>
<td>256-tap IFFT</td>
<td>2,585</td>
<td>11</td>
<td>42</td>
</tr>
<tr>
<td>θ15</td>
<td>64-word Cyclic Prefix Inserter</td>
<td>903</td>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>θ16</td>
<td>OQPSK Modulator</td>
<td>596</td>
<td>11</td>
<td>70</td>
</tr>
<tr>
<td>θ17</td>
<td>PSD Estimator</td>
<td>1,573</td>
<td>11</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 9.3: Task-set in the SDR prototype

The three communication standards implemented by our SDR prototype as well as our cognitive radio solution were built using the aforementioned task-set. The tasks are executed in a different order and with some minor changes in each case, as shown in the Directed Acyclic Graphs (DAGs) in Figures 9.10 (cognitive radio), 9.11 (WiFi), 9.12 (WiMAX), 9.13 (UMTS). For instance, the generation polynomial used in the convolutional encoder is different in WiMAX and UMTS standards, and the interleaving pattern and length changes
from one standard to the other. Therefore, the tasks were parameterised to allow small online adjustments to be made in order to fulfil the requirements of each communication standard. In order to ease this process the adjustable parameters were mapped to directly accessible FPGA resources, such as LUTs. On the other hand, different versions of the same task were developed when the amount of circuitry varies significantly with the values of the adjustable parameters. For instance, $\theta_{14}$ is capable of implementing both 64 and 256 IFFT taps, but consumes a notably larger amount of FPGA resources than $\theta_{11}$, which is specifically designed to compute 64-tap IFFT. In order to enable $\theta_{14}$’s reuse to compute 64-tap IFFTs, the size of the IFFT is kept as a parameter in this task. If $\theta_{14}$ is not already configured when a 64-tap IFFT is required, the system will proceed to allocate $\theta_{11}$, which takes less time and consumes less FPGA resources.

Xilinx System Generator tool was used to translate the high-level task models into synthesisable VHDL code, which was in turn adapted to the partial reconfiguration design flow, i.e., non-reconfigurable resources were extracted (DCM, BUFG, etc.). The modified VHDL was then combined with R3TOS specific logic (e.g., TCL) and placement constraints (e.g., PRRs definition) prior to synthesising it using XST to obtain the partial bitstreams of the tasks. Finally, the latter bitstreams were manipulated as required by R3TOS and loaded to the external memory included in the R3TOS prototype.

An important point to be considered in our SDR application is the great difference between the input and output throughput of some of the tasks. For instance, the WCDMA modulator used in cognitive radio ($\theta_8$) generates 1 Kb at its output per each input bit due to the spread spectrum technique used by UMTS. Since the size of the data buffers of the tasks is limited by the availability of BRAMs in the device, tasks with high output-input throughput relation need to execute several times to process all the input data, i.e., the remaining input data in the IDB after each task execution must be processed in the next execution. In the following we will refer to these tasks as reentrant. In Figures 9.10, 9.11, 9.12, and 9.13, the size of the data buffers of the tasks as well as the number of times each reentrant task needs to execute are shown.

In general, the execution time of SDR tasks is deterministic. It depends on the amount of data to be processed, the input interface width, and the relation between input and output clock rates. However, the different combinations of variable data produced by the tasks
Figure 9.10: Cognitive radio: DAG
Figure 9.11: WiFi DAG
Figure 9.12: WiMAX: DAG
Figure 9.13: UMTS: DAG
which operate in windows, i.e., the remaining data in the tasks’ buffers are accumulated for several executions until they eventually form another data window, results in data transfers of different lengths with the D/A controller. This makes the whole system functioning complex, being necessary to use queueing theory [Allen, 1990] to gain a detailed knowledge of it. This is indeed mandatory to prevent memory overflows in the data buffers.

In connection with the above concept, an important aspect to be considered in order to approach real-time is how to model data transfers between hardware tasks and system input/outputs, which indeed are carried out through the ICAP While input transfers (e.g., with the CIF image sensor and A/D controller) are included in the data delivery phase of the tasks, output transfers (e.g., with the D/A controller) are not considered in any task. In order to tackle this issue, we resorted to create a communication specific task, $\theta_{18}$. This task is conceptually similar to that presented in chapter 4 in the scope of hardware-software communications: it does not consume any on-chip resources and consists only of a set-up phase during which data is transferred (i.e., execution time is equal to zero).

9.3.2 Application Functioning and Experimental Evaluation

At runtime, the order of execution of the tasks is dictated by the application running on the main CPU. Besides, the main CPU should execute a set of software routines to manage data windows and reentrant tasks. These routines, which are not implemented yet, should work in conjunction with some kind of hardware support added to the TCLs to abstract the reconfigurable application’s functioning to the user. More specifically, they should keep track of the amount of data stored in the task buffers, and based on these values, they should trigger the appropriate data consumer tasks as many times as needed and with the suitable parameters (e.g., $t_{E,i}$, $t_{D,i}$, $t_{R,i}$).

Task allocation is especially important in this application in light of optimising data transfers among successive SDR tasks in the DAG and promoting the reuse of already configured circuitry in the chip. In line with these two objectives, data producer and consumer tasks are packaged in pairs within the same clock region most of the times. On the other hand, since the number of tasks which are ready at the same time is limited, task scheduling is not so important in this application. An important issue to be noted here is the limited allocatability of heterogeneous tasks due to the layout of the used chip. Namely, only two
SDR tasks can be packaged together into the same clock region as there are only three heterogeneous resource columns in the device (e.g., BRAM and DSP48).

It is significant to note that the task preemption mechanism explained in chapter 7 was successfully used with the data randomiser task ($\theta_2$). This task includes a Pseudo Random Binary Sequence generator (PRBS) to implement simple data encryption; i.e., the pseudo random sequence generated by the PRBS is XORed with the message data. In order to ensure coherent data recovery in the receiver, the PRBS must continue generating the pseudo random sequence from the point it stopped in the last execution. This is why it is necessary to save the context of this task; i.e., the state of all of the registers.

We report basic functioning of the SDR transmitter prototype. Namely, the prototype was able to transmit data when required, circumventing injected interferences in the transmission channel. However, there are many application-dependent issues which have not been adequately addressed yet (e.g., management of windows and reentrant tasks), resulting in failure to correctly receive the transmitted images. The data buffers of the tasks were used as test points to check the correct functioning of our prototype. In effect, the content of these buffers was read-back and compared against the data produced in the System Generator simulation environment (i.e., MATLAB/Simulink). We checked that the data sent to the RF front-end was correct when processing the first data windows, but as expected, it turned wrong when executing reentrant tasks successive times.

Besides, we report large time overheads in our prototype due to the fact that the execution phases of most SDR tasks are shorter than their set-up phases ($t_{E,i} < t_{ICAP,i}$). As a result of these overheads, the time needed to transmit an image ranged from less than a second (in WiFi and WiMAX) up to some seconds (in UMTS and cognitive radio), which is not sufficient for most applications (e.g., real-time video streaming).

As an ending note, we would like to emphasise that this case-study is aimed at showing the feasibility of using R3TOS for the rapid development of reconfigurable applications. Most of the aspects which play an important role in the achievable performance have been neglected (e.g., codesign and application partition into tasks) and should certainly be revisited in a next stage of research. For instance, SDR hardware tasks could be based on a small processor (e.g., PicoBlaze) coupled with custom logic instead of relying exclusively on cus-
Chapter 9 - R3TOS Prototyping and Application Development

tom hardware to perform the computation. By doing so, a more effective trade-off between resource usage (task size would be smaller) and execution time (it would be longer) could be achieved. As task set-up times would be likely to be smaller than execution times, the situation where R3TOS is expected to achieve better results could be approached. We believe that although the execution speed of individual computations would decrease, the overall performance of applications with sufficiently large computing demands could be improved due to better exploitation of the multitasking capabilities delivered by the FPGA, i.e., the device could be kept more occupied with tasks performing active computation for longer periods of time. This scenario is interesting as other functionality could be executed using the saved on-chip resources, being thus possible to target more sophisticated applications.

9.4 Chapter Conclusion

This chapter firstly presented a proof-of-concept implementation of R3TOS using real Xilinx FPGAs, and then shown the feasibility of using this implementation as a platform to develop two real-world applications: a traction controller oriented to railway transport and a cognitive SDR transmitter. While the former application was used to demonstrate the improved reliability achievable with R3TOS, the latter application demonstrates the feasibility of using R3TOS for the rapid development of reconfigurable applications, working in conjunction with currently available high-level design tools; e.g., Xilinx System Generator. However, the current SDR implementation has manifested limitations to deal with data-window processing and fails to achieve high-performance, showing the key importance of making the correct design decisions (e.g., size of data buffers and codesign). Better results are expected when using R3TOS in applications with a high computation to allocation ratio.

Finally, we would like to point out that, at the time of writing this thesis, R3TOS is being also used to develop a K-means clustering application based on the work presented in [Hussain et al., 2011], a pairwise biological sequence alignment application based on the work reported in [Isa et al., 2012], and a cryptographic solution based on the work presented in [Adi and Benkrid, 2010]. A significant performance improvement has been shown when comparing our R3TOS-based implementation of the former two applications, which exploit the task clonation capability, with entirely software implementations. In addition, preliminary results obtained in the pairwise alignment application, where the number of
processing elements included in the tasks (i.e., task size) can be adapted online, show a
threefold improvement over the situation where no runtime task adjustment is done [Hong
et al., 2013]. This is due to a more efficient use of FPGA resources. Future work will aim to
characterise R3TOS in terms of performance and efficiency in the scope of these K-means,
pairwise alignment and cryptographic applications as well as many others.
Conclusions and Future Work

“We can only see a short distance ahead, but we can see plenty there that needs to be done.”
—Alan Turing

This thesis has introduced a new solution (R3TOS) to deal with some of the major challenges that are arising in the area of reconfigurable computing, namely to develop fault-tolerant, high-performance, real-time, low-power and adaptable reconfigurable computing solutions from high-level descriptions.

R3TOS provides systematic OS-like support to reconfigurable hardware (i.e., FPGAs), easing the exploitation of some of the most advanced capabilities of this technology by unexperienced users. Indeed, by wrapping reconfigurable hardware with a real-time software microkernel (e.g., FreeRTOS), R3TOS creates a unified hardware-software runtime execution environment, whereby a software-centric application developer can easily use the underlying hardware resources to reliably benefit from increased computing speed with smaller power consumption than achievable when using a conventional processor. In this context, R3TOS can be seen in multiple ways beyond an OS: as an abstraction model; as an interface; but above all, as an enabling solution towards mainstream usage of reconfigurable hardware.
Chapter 10 - Conclusions and Future Work

Notably, R3TOS is not an isolated proposal. It must be viewed in line with on-going efforts to develop new high-level design tools for hardware as well as new programming styles and languages to integrate parallelism and ease multitask decomposition. Eventually, automatic implementations of hardware and software from higher-level descriptions will be the only way to deal with the ever growing complexities and design costs of modern computing platforms [Schirrmeister, 2012]. R3TOS is to fit in the heterogeneous computing paradigm of the future as a runtime support provider for the hardware modules (i.e., hardware tasks), and software routines (i.e., software tasks), that are to be targeted to modern FPGA platforms, interacting on a peer-to-peer relationship. In this chapter, we present final conclusions and plans for future work. Before that, the next section summarises the main achievement of this thesis in light of the objectives introduced in chapter 1.

10.1 Summary

What makes R3TOS special is its non-conventional way of exploiting chip resources: these are used indistinctly for carrying out either computation or communication at different times. Indeed, R3TOS does not rely on any static infrastructure apart from its own core circuitry, which is constrained to a specific region. Thus, the rest of the device is kept free of obstacles, with the resources ready to be used when needed. In this context, the partial bitstream relocation technique has been enhanced to allow hardware tasks to be mapped to different chip locations on-the-fly, with the dual objective of improving computation density (i.e., keep the tasks as compact as possible), and circumvent damaged resources. Note that permanent damage may be caused by imperfections in the fabrication process or it may emerge as the silicon ages.

R3TOS is able to exploit locality by implementing multi-rate clocking capability. Unlike in an ordinary FPGA implementation, where data is moved among resources which may be potentially located anywhere on the device, in R3TOS data is kept within a much smaller physical area (i.e., within hardware task's boundaries), resulting in shorter paths and smaller access times. R3TOS automatically takes advantage of the reduced access times as it feeds each task with its maximum allowed clock rate to obtain the highest performance from each portion of the design. If required, the clock frequency delivered to each task can be dynamically chosen with the objective not to increase performance, but to reduce the power con-
sumption and heat dissipation. In this line, dynamic power consumption is automatically reduced by clocking resources locally (i.e., the resources which are not clocked only consume a leakage current). Note that by doing this, unnecessary wear of the resources which are not in use is also prevented.

In R3TOS, locality is combined with global communications. Indeed, input data is received in one of the edges of the hardware tasks, it is processed as it flows through tasks' data-path and, the final results are stored in another edge of the tasks. This promotes the concatenation of tasks, as the input edge of one task can be placed next to the output edge of the previously executed task in the pipeline. Thus, there is no specific circuitry dedicated to communicate data through the chip. Data is automatically moved as a consequence of computing in space, i.e., data moves locally inside each task, and globally from task to task. When task concatenation is not possible, two alternative communication methods are envisioned. First, on-demand interconnection wires can be created using the FPGA resources - these wires only remain configured while data is transferred between tasks. Second, the FPGA's reconfiguration mechanism can be harnessed to relocate data from task to task without using any physical support. By using any of these methods, R3TOS is always able to provide a logical interface for the tasks, regardless of their physical location on the chip, enabling inter-task communications and synchronisation.

R3TOS also permits to approach the implementation of Fault-Containment Units (FCUs), which are necessary to achieve higher levels of dependability [Lala and Harper, 1994, Kopetz, 2011]. Indeed, as hardware tasks work in isolation on FPGA, i.e., chip resources are not shared among tasks at any particular time, any single fault (excluding power and clock supply failures) affects only one task and does not propagate to other tasks. In order to tolerate and detect faults, R3TOS executes redundant instances of critical tasks. When a task instance comes up with a different set of results, or it does not meet the expected timing, that instance is marked as erroneous and a diagnostic test is executed to find out the source of the fault, i.e., localise damaged resources within the region assigned to the task, if any. Notably, due to the dynamic nature of the tasks, as they are continuously reconfigured on the device, soft-errors do not accumulate as they are automatically scrubbed. Likewise, FPGA device's lifetime is expected to prolong as a result of distributing the implemented functionality along the entire chip over time, which would leverage the wear.
Note that most of the capabilities offered by R3TOS are not directly supported by any current standard design tool, and it has not been until some months ago that Xilinx Inc. has launched the Isolation Design Flow (IDF) [Corbett, 2012] to give support for implementing FCUs on FPGA-based static systems.

Last but not least, R3TOS is to ease the development and certification of reconfigurable systems. The support given by R3TOS allows designers to develop applications with relatively arbitrary interfaces. Besides, once the application-independent core circuitry of R3TOS is certified, the application-dependent tasks can be individually designed and verified (i.e., component-based design), based on the fact that their interactions are safely carried out by R3TOS. Inspired by software OS and aiming at approaching certification, R3TOS implements a privileged access to the shared FPGA’s configuration memory to protect the configuration state of the system from being corrupted when carrying out undesired accesses. Being a single source of failure, R3TOS circuitry has been carefully protected against faults, triplicating the critical logic and implementing a set of self-diagnostic mechanisms. Despite the resulting area overheads, the total amount of FPGA resources used by the R3TOS logic is kept reasonably low. As an additional contribution in this line, a fault-tolerant implementation of the popular Xilinx 8-bit PicoBlaze processor (KCPSM3-FT) has been developed. The latter can be very useful for a wide variety of small designs that require high reliability, such as the system proposed in [Reibel Boesen, 2011].

10.2 Reflections

The single most important fact about R3TOS is that it does work. This in its own right is a proof that the main objective of this thesis has been met. Beyond the primary conclusion that R3TOS works, the statements and arguments made about it in previous chapters, and mentioned again earlier in this chapter, are next revisited. Namely, it has been argued that R3TOS allows for exploiting real-time, parallel and multitasking capabilities of pure hardware as well as approaching more dependable, adaptable and efficient systems, which can be programmed at a higher-level of abstraction. Additionally, lower power consumption has been also claimed for R3TOS-based reconfigurable systems.

Our final conclusions are mainly drawn from our own experience when developing and evaluating the demonstrative applications described in chapter 9, namely Software Defined
Radio (SDR) prototype and railway traction motor controller, as well as other applications that are still under development at the time of writing this thesis. We note that, besides being the designers of R3TOS, we have become the very first users of R3TOS as well.

The section is concluded with a summary of the most important open issues we currently see.

**Real-Time**

The developed demonstrative applications make limited use of real-time. It has been proved that R3TOS is capable of (1) correctly generating digital signals with strict timing requirements (i.e., PWM pulses), (2) coordinating the access of various tasks to FPGA resources for keeping a continuous video stream transmission, and (3) detecting timing violations (i.e., missed deadlines). However, the capability to schedule a set of concurrent real-time computations has not been tested beyond the simulation evaluation presented in chapter 5. In any case, we note the realism of our simulations which consider most of the limitations of current FPGA technology (e.g., resource heterogeneity and ICAP sequentialness), as well as major particularities of reconfigurable computation (e.g., inter-task dependencies and usable clock frequency). We finally point out that the use of our algorithms in hard real-time safety critical applications is not allowed, as there is no feasibility test that guarantees no deadline will be missed. Instead, real-time capability of R3TOS is aimed at guaranteeing a good Quality of Service (QoS).

**Multitasking and Parallelism**

In the developed demonstrative applications, the use of parallelism is more oriented towards reliability (i.e., redundant instances of the PC-PWM\textsuperscript{1} circuit run concurrently) rather than performance (i.e., most of SDR tasks are sequential). Unfortunately, the multitasking capabilities of R3TOS are limited due to the sequential bottleneck introduced by the reconfiguration mechanism of current FPGAs (i.e., ICAP). Notably, R3TOS uses a custom-designed ICAP driver that is able to approach the maximum theoretical speed of the ICAP for the Virtex-4 family, thus alleviating the reconfiguration bottleneck. Furthermore, R3TOS uses bitstream compression (i.e., MFWR commands) to speed-up the allocation of multiple cloned task instances as well as to deliver to them the same set of input data (e.g., measured

\textsuperscript{1}PC-PWM is the Proportional Control PWM block used in the traction motor controller.
train speed and target speed in the traction controller application). In the scope of the K-means application we are currently developing, the latter technique is used to enhance multitasking by rapidly configuring multiple identical K-nearest neighbour classifiers [Ebrahim et al., 2013], which are provided with a different set of data each, possibly, coming from multiple user requests. In this application, preliminary results using clonation show a more than twofold speed-up. Likewise, in the pairwise sequence alignment application, clonation is used to configure the required number of pipelined processing elements inside each task. However, this technique cannot be used when the tasks are not identical, resulting in significant time overheads in the SDR prototype, where the execution phase of the tasks is too short with regard to their set-up phase. Overheads are to reduce when the tasks execute for a period of time that is sensibly longer than the amount of time needed to configure them on chip. Therefore, multitasking in R3TOS is highly dependent on the partitioning of the reconfigurable application (i.e., set-up to execution time ratio of the tasks) and, as occurs with most technologies, the range of applications that can benefit from R3TOS multitasking is constrained.

**Dependability**

The improved dependability offered by R3TOS has been demonstrated in the scope of the traction motor control application. The results obtained when injecting faults to our R3TOS-based traction controller confirm that its failure rate is significantly smaller than that of an equivalent controller which uses TMR (up to 85% less failures were detected when the fault injection rate was considerably high, i.e., 32 faults per second). However, since the soft-errors and permanent damage occurrences were physically simulated by synthetic random data rather than real-world data, a slight deviation might be expected when using R3TOS in truly harsh environments. Hence, it should be convenient to carry out a more exhaustive dependability assessment of R3TOS (e.g., using a cyclotron).

**High-Performance**

Besides the already mentioned use of bitstream compression to cope with the reconfiguration bottleneck of FPGAs, R3TOS is able to increase overall performance by reusing both intermediate partial results between different computation stages and previously configured pieces of circuitry on the device. Inter-task communications are also accelerated by
exploiting the vast matrix of programmable wires that offer an outstanding capability for moving data among registers and memory elements. Another mechanism of R3TOS to increase the performance is its multi-rate clocking scheme that approaches the possibility that each task run at its maximum speed. These improvements are collectively included in the Snake task allocation strategy [Iturbe et al., 2011d]. However, the performance drastically declines when dealing with low computation to communication ratios (e.g., HBC tasks) if it is not possible to form large chains of interconnected tasks (e.g., when the FPGA is very occupied/damaged or it includes few BRAM columns). Finally, it is important to note that beyond the specific techniques deployed by R3TOS to gain performance, this is expected to be high because of the computation specialisation and parallelism provided by the hardware tasks. That said, the achievable performance improvement ultimately depends on a number of factors which are out of the scope of R3TOS, including the amenability of the computation to be accelerated by hardware, the effectiveness of application decomposition into tasks and the quality of synthesis tools.

**High-Level Programming**

The fact that our SDR application was programmed in a software environment, i.e., Xilinx Software Development Kit (SDK), demonstrates the feasibility of accessing FPGA resources using R3TOS-based traditional software-like APIs. As we did not have to be aware of low-level FPGA details when programming our application, and following the terminology used in software OS, we conclude that R3TOS allows for successfully virtualising FPGA resources. Furthermore, as the SDR hardware tasks were implemented using currently existing high-level hardware design tools, i.e., Xilinx System Generator, we note that we enjoyed a nearly complete high-level programming experience. The only remarkable limitation was the fact that currently there is no automated environment for developing R3TOS-based applications. For instance, the Task Control Logic (TCL) had to be manually attached to the tasks and the tasks’ bitstreams had to be manually loaded to the bitstream memory.

**Efficiency**

The efficiency of R3TOS-based applications, also in terms of power consumption, is sustained in the fact that hardware is always more efficient than software [Compton and Hauck, 2002, Tian and Benkrid, 2010, Prasanna Sundarajan, 2010, Tse, 2012]. Indeed, hardware
specialisation is thought to be the only way to go beyond the power wall. Besides, R3TOS makes a more efficient use of FPGA resources than most of related reconfigurable systems as it exploits slotless reconfiguration [Hong et al., 2013]. However, it introduces some overheads with regard to bare hardware, namely the chip resources used to implement R3TOS engine, thus reducing the amount of usable resources by application-dependent hardware tasks. This overhead is less than 10% of the total slices in an XC4VLX160 part, notably with no resources required to implement static communication infrastructure. R3TOS also results in time overheads by virtue of its own functioning time. Despite the latter overheads are minimised by parallelising most of R3TOS functioning with ICAP-access times, they are still in the range of tens of microseconds per each task allocation attempt, with a greater value when the software-like API is used. As a result of time and area overheads, application execution is slower and power consumption increases. We posit that there are many applications that can accept this efficiency loss in exchange for the higher productivity and adaptation offered by R3TOS. Indeed, the current competitive market is clearly not asking for the highest efficiency, if it comes at the price of longer development times. Perhaps it is difficult to justify R3TOS for performance-conscious hardware designers, who usually work in the nanosecond range, but there are plenty of arguments in its favour for productivity-conscious system developers, who indeed dominate the current application development landscape and usually work in the millisecond range.

**Adaptability**

The adaptability delivered by R3TOS is shown in the case-study prototypes reported in chapter 9: (1) the R3TOS-based traction motor controller is able to adapt its own architecture to circumvent emerging damage in the device and, (2) the SDR prototype is capable for adapting its functioning. More specifically, it changes the communication standard and transmission frequency based on the availability of networks and spectral bands at each time. Beyond the case studies presented in this thesis, adaptability is being developed in the scope of the aforementioned biological pairwise alignment application, where the number of processing elements included in the tasks can be dynamically modified based on the availability of FPGA resources and the computation demands at each time [Hong et al., 2013].
10.3 A Final Critique of R3TOS

From a critical perspective, R3TOS moves the von Neumann bottleneck from the program memory of a processor to the configuration memory of an FPGA. Indeed, the reconfiguration bottleneck is provoked by the ICAP, which is nothing more than the interface to the internal configuration memory of the FPGA. Despite the memory bandwidth limits the performance in both cases, it is more restrictive in the case of a processor, where the computing power depends on the number of fetched-executed instructions. On the other hand, although the relation between the amount of time needed to configure and execute a hardware task depends on the type of computation it implements, in general there is a trade-off here: a task can be kept smaller (i.e., faster to allocate) by reducing the data parallelism and the pipeline depth, usually resulting in longer execution time. Here lies the benefit of R3TOS. Furthermore, it is important to note that R3TOS includes a set of techniques to cope with the configuration memory bottleneck, including the use of bitstream compression, to configure multiple cloned instances of the tasks, and the reuse of already configured hardware tasks and intermediate partial results computed by them.

Like in software, another important trade-off in R3TOS is the computation to communication ratio. It is thus of utmost importance to efficiently partition an application into tasks. The objective here is to exploit at maximum the capability to rapidly and simultaneously transfer data among the registers mapped into the same tasks (i.e., RTL communications), and reduce inter-task serial communications that penalise overall performance. Note that as FPGAs evolve and become larger, they will be able to allocate larger tasks with presumably more RTL communications inside.

A key requirement to harnessing FPGA's computational power is to successfully hide the latency of ICAP with computations. R3TOS is to accomplish this by taking advantage of multitasking when making the scheduling decisions: while one task is configured, other tasks can perform active computations, thus increasing overall FPGA utilisation and hiding ICAP latency. Note that in order to do so, the tasks with shortest set-up phase and longest execution phase should be first scheduled. However, we note that harnessing the whole FPGAs computing power conflicts with real-time, which indeed requires to modify the order of execution of the tasks to meet the computation deadlines. Currently, R3TOS prioritises
real-time over performance, but this could be easily changed by just modifying the used scheduling algorithm. Despite not having been conveniently explored in this thesis, the performance loss due to the ICAP latency is expected to be minimised by exploiting software multitasking as well: while one hardware task is configured, the main CPU can perform active software computation.

In light of the above, it can be concluded that R3TOS opens new opportunities for computing, even if it is not trivial to take advantage of them.

However, we acknowledge that current reconfigurable technologies, and more specifically FPGAs, have not been designed to be used as proposed by R3TOS and similar approaches. Despite the fact that the configuration memory of Xilinx FPGAs can be accessed at runtime, it is definitely not designed to allow high throughputs. Unfortunately, as the reconfiguration speed provided by the reconfiguration mechanism increases, the time overheads provoked by R3TOS are to be more notorious. In any case, we report there is still significant margin for reconfiguration speed improvement without affecting performance, i.e., while R3TOS overheads are measured in the range of tens of microseconds, the time needed to set-up a typical hardware task is in the range of hundreds of microseconds. Moreover, some of the operations defined in R3TOS are not currently fully supported (e.g., task context switch), or can be done with restrictions (e.g., task reallocation). In case our proposal is accepted by a significant amount of people, we hope this will change in the future, allowing us to implement our ideas more easily and efficiently. For instance, the ICAP bottleneck could be relaxed if multiple configuration ports were available, and task relocatability would be substantially improved if the chip layout was more homogeneous; i.e., same number of CLBs between BRAM/DSP48 columns. Following this list of desirable claims for FPGA manufacturers, we highlight the convenience of increasing the capacity of BRAMs. Indeed, there is an inefficiency due to the different proportion of logic and memory resources in current FPGAs (i.e., CLBs to BRAMs). Small tasks that process high amounts of data must often be enlarged in order to include the necessary amount of storage resources. In the way towards FPGA universalisation that R3TOS is intended to follow, we note that the FPGA fabric should be improved to address the needs of new types of computations. For instance, better floating-point support should be developed [Craven and Athanas, 2007, de Dinechin et al., 2008].
Finally, we acknowledge that the success of R3TOS to cope with permanent damage emerging on the chip is conditioned by the effectiveness of the used diagnostic mechanism. Despite the currently proposed mechanism in this thesis has not been tested with real partially damaged FPGAs, R3TOS is conceptually opened to work in conjunction with most of research efforts in the field, such as Built-In-Self-Tests (BISTs).

10.4 Upcoming Technologies and Future Perspectives

R3TOS is entirely dependent on the evolution of reconfigurable technology, especially on the improvement of reconfiguration speed. Nonetheless, we have reasons to be optimistic about this. Indeed, when compared to the bandwidth offered by modern memories, there is still a margin for improvement. Currently, the officially reported reconfiguration speed for Xilinx FPGAs is only 400 MB/s and the highest reported speed is 2.2 GB/s [Shelburne et al., 2008, Bonamy et al., 2012], which is still far away from the bandwidth offered by brand generation of DDR4 memories (i.e., 21 GB/s [Shilov, 2012]).

But a serialised reconfiguration port is not the best option for a device able to perform concurrent computations in space. To tackle this limitation, Optically Reconfigurable Gate Arrays (ORGAs) have been proposed in the last decade [Nakajima and Watanabe, 2009]. ORGAs extend parallelism to the reconfiguration process with the objective of circumventing the reconfiguration bottleneck of current FPGAs. The new technology is to enable rapid reconfiguration by using a holographic memory and optical wide-band reconfiguration connections, leading to negligible configuration overheads (in the nanosecond-order), and enhanced multitasking capabilities. Thus, the next base technology for R3TOS might be ready in the technology roadmap.

Being very aligned with R3TOS, a technology that has already moved from laboratories to the real-world is 3-D Programmable Logic Devices (3PLDs), such as Tabula’s Spacetime 3D architecture [Morris, 2010]. In fact, this technology follows up research efforts begun as early as 1994 to build a multi-context dynamically reconfigurable FPGA [DeHon, 1994, Trimberger et al., 1997]. A Spacetime device reconfigures on-the-fly at multi-GHz rates, executing each portion of a design in a sequence of steps. Each step defines a specific circuitry that takes the results computed by the preceding step from pre-defined positions, processes them, and leaves the results in other pre-defined locations, ready for the next
step. This differs from R3TOS, where computations steps are scheduled online and data exchanges occur at arbitrary positions. While Spacetime technology appears to be promising, especially because of the high reconfiguration speed it achieves, it suffers from the same limitations as off-the-shelf partially reconfigurable FPGAs, that is, the lack of suitable tools to develop reconfigurable applications.

Finally, another reality at the time of writing this thesis is the new family of Xilinx FPGAs Zynq, which include an ARM dual-core Cortex. Allowing to merge both hardware and software worlds, we envision that this family of partially reconfigurable FPGAs should be the base technology for next R3TOS releases until the upcoming generations of reconfigurable hardware described above push the market.

In the context of high-level hardware design tools, we note that Xilinx Inc. has recently made available the Vivado Design Suite, which enables the direct translation of C, C++ and SystemC specifications into FPGA implementation without the need to manually create RTL. Namely, Vivado provides support for arbitrary precision bit-width data types, unrolling and pipelining iterative loops, scheduling memory accesses and implementing custom interfaces (e.g., bus, registers, FIFO, or RAM). Likewise, Altera Corp. has developed OpenCL SDK tool to implement OpenCL applications on FPGAs. While initiatives such as Vivado and OpenCL SDK are very important to universalise the use of reconfigurable hardware, R3TOS complements them by providing high-level runtime support to universalise the exploitation of dynamic reconfiguration opportunities.

10.5 Future Work

This thesis has set the basis and proved the feasibility of R3TOS, but further work needs to be done to build a more efficient, optimised, powerful, and in short, a mature version of it.

One of the major weaknesses of our solution is the lack of a unified development framework to specify R3TOS implementation and functioning directives as well as to automate the set-up of R3TOS-based reconfigurable applications. Currently this is a tedious process that must be manually done by the user using standard design tools. The reason why this work has not been completed until now is that it is not central to prove the concept of R3TOS. However, in order to promote the adoption of R3TOS by industry and academia, an automated development framework such as the one presented in [Bonetto et al., 2012] should
be built in the future. This framework should guide the developer from the original partitioned application, described as a task graph, down to its deployment onto the target device.

An important issue to work on is the integration of R3TOS, and more specifically, the loose integration of the hardware low-level services into the software microkernel. As pointed out in chapter 7, this integration should occur in a more natural way. The services implemented by the software microkernel should directly invoke the services implemented by the hardware microkernel. Moreover, it must be noted that R3TOS API should evolve with the necessities of application developers. In the future it might well be that the software microkernel is to be replaced by a new trendy alternative. In this line, it is worth saying that the next planned release of R3TOS is to use the ARM core included in the new Xilinx Zynq family of FPGAs, for which more standard software solutions are available. Besides, other APIs commonly used in high-performance parallel-computing should be explored in the future (e.g., CUDA or OpenCL).

From an algorithmic and theoretic perspective, more precise models of reconfigurable computing should be developed in the way to building suitable tools for designing reconfigurable applications, including hardware-software multitask partitioners. An important problem here is that reconfigurable technology is constantly developing further, and thus the limitations and particularities that should be addressed by the aforementioned theoretic computation models also change quickly. Having said this, a schedulability test for real-time reconfigurable computing would be extremely useful, and the development of lighter scheduling and allocation algorithms might well be mandatory as the reconfiguration speed of reconfigurable technology increases.

Scaling and extending R3TOS as reconfigurable technology evolves and FPGAs become faster and more complex is going to be an exciting research topic. On the one hand, using larger FPGAs with shorter reconfiguration times will make systems more powerful, but on the other hand, longer time will be presumably needed to make the allocation decisions in the new devices. Nonetheless, we note that R3TOS is argumentatively better positioned to face this issue than related approaches. First, Snake is able to exploit locality and thus is less dependent on chip size. Second, the computation of EAC and EVC allocation heuristics, which do depend on the chip size, can be accelerated either by relying on a more parallelised hardware implementation (incurring higher area overheads), or by trading-off efficiency in
the management of FPGA resources, i.e., faster executions can be achieved by increasing the managed granularity in the chip. Finally, the fact that R3TOS has been designed as a microkernel, with separate servers to implement the ROS processes, is thought to ease its further development.

From a technological perspective, research efforts should be focused on improving the partial bitstream relocation technique (e.g., deal with resource heterogeneity). However, new opportunities and challenges that we cannot predict now will certainly appear as FPGA technology evolves.

A central limitation when working at low-level with FPGAs is the lack of information provided by manufacturers. For instance, there is little and incomplete information about the format of the bitstream of Xilinx FPGAs. We have obtained this information using reverse engineering for the specific case of Virtex-4 FPGAs, which was the technology adopted in this thesis. Unfortunately, this tedious process, which took a considerable amount of time during our research, must be repeated for each new generation of FPGAs, making slower the development of R3TOS and discouraging the proliferation of similar initiatives. However, we note that future releases of R3TOS should target more modern devices (e.g., Virtex-6 and Virtex-7), as Virtex-4 is becoming obsolete.

Besides, new capabilities should be added to R3TOS. For instance, the dynamically reconfigurable temperature sensors presented in [Lopez-Buedo and Boemo, 2004] could be swapped in and out on the FPGA to monitor the heat dissipated on each region. This information could be used by R3TOS to better leverage the wear when making the allocation decisions.

In the scope of reliability, it is mandatory to include a proven damage detection diagnostic mechanism. Furthermore, an interesting capability to be added to R3TOS is self-replication [Tempesti et al., 2007]. Ideally, R3TOS should be able to relocate its own circuitry to another location within the FPGA when it fails. The main challenge here is how to maintain the connections between the relocated circuitry and the fixed FPGA resources, including I/O pins, Frame ECC, STARTUP logic, and especially, ICAP port, through which the replicated R3TOS instance is to be configured.
An interesting capability to be explored in the scope of multitasking is how to force interruptions on R3TOS from the hardware tasks. Currently, the computation completion of the tasks is checked by polling their hardware semaphores and hence, this unnecessary overhead could be removed if the tasks were capable to signal their finishing. We have been unable to figure out a way to achieve this functioning in current Virtex-4 FPGAs, beyond relying on wires to connect the tasks with the R3TOS circuitry. Note that the latter requires online routing capability and might result in allocatability problems for the rest of the tasks. Indeed, the only way we envision to reach any logic resource on the FPGA without resorting on online routes is using the clock-tree, but this is unidirectional and limited, as it only connects the logic and clocking resources. Besides, in order to improve the performance and multitasking capabilities, all of the concatenated tasks in a chain should be able to perform computation simultaneously; i.e., producer and consumer tasks should be able to access shared BRAMs at the same time. In order to achieve this, it should be investigated a method to provide the tasks with “standard” BRAM interconnections in light of making them compatible with each other.

Despite being against the original foundations of R3TOS, we acknowledge that including a static communication infrastructure is convenient in the scope of high data rate applications (e.g., multimedia). Namely, a static infrastructure is a good alternative to achieve high bandwidth communications when the (high data rate) tasks cannot be connected with each other, i.e., Snake cannot be used. However, as we have pointed out in this thesis, using a static communication infrastructure would lead to the unresolved issue of having to deal with static routes across the chip. Having said this, we note that specific solutions can be found for specific chip layouts. For instance, in [Iturbe et al., 2010a] we propose to implement a NoC using the BRAMs located in the edges of the Virtex-4 XC4VLX160 device, thus leaving the central sandbox free of static routes to allocate the hardware tasks. Namely, high data rate tasks are to be allocated close to the NoC interfaces, while low data rate tasks are to be preferably allocated on the opposite part of the sandbox. However, despite this allocation scheme allows the efficient sharing of FPGA resources between both types of tasks, its applicability is limited in modern and heterogeneous devices. In the worst case, each type of task could be allocated on a different half of the FPGA, using a communication infrastructure to interconnect high data rate tasks on the corresponding half.
In the scope of signal processing applications, new TCLs with capability to deal with data windows and reentrant tasks should be developed. Ideally, the high-level synthesis tools that are currently pushing the market, such as Xilinx Vivado, should be provided with capability to generate R3TOS TCL-based interfaces.

Finally, a “killer” application for R3TOS should be found in order to fully show the extent of its possibilities and thus attract potential users. After all, as learned from the software field, a successful OS is always accompanied by an attractive and powerful set of applications for it, see the case of Microsoft Windows.
Design and Implementation of a Fault-Tolerant PicoBlaze: KCPSM3-FT

Being the basic processing core in R3TOS HWuK, PicoBlaze is also the most prone to failure. This vulnerability mainly comes from the BRAM(s) from which PicoBlaze runs its program. Indeed, in the Xilinx Virtex-4 family of FPGAs, BRAMs are about two times more susceptible to faults than any other logic resources, with 484 FIT/Mb compared to 263 FIT/Mb for the other resources [Xilinx Inc., 2012a].

As mentioned in chapter 2, Xilinx Inc. has developed an ECC circuitry that can be used in conjunction with two vertically adjacent BRAMs to make a 512x64-bit memory with built-in Hamming error correction (i.e., RAMB32_S64_ECC). The ECC circuit permits to correct single-bit errors and detect (but not correct) double bit errors occurring in the memory data words. While Xilinx Inc. has recently provided support for using ECC-protected BRAMs to store the code program executed by 32-bit MicroBlaze soft-cores and PowerPC hard-cores, there is currently no support for doing the same with an 8-bit PicoBlaze.

In order to enable this capability in the latter processors, we have designed an adaptation logic, i.e., ECC Processor Adapter (EPA), that makes PicoBlaze and ECC-protected BRAMs compatible. EPA synchronises processor instruction fetching and ECC-protected BRAM read. Moreover, the EPA also includes a scrubbing logic which is coupled to the ECC-protected program BRAM to fix single-bit errors without affecting processor execution. When non-correctable multiple errors are detected in the program memory, a full FPGA re-
configuration is forced as described in chapter 4 (i.e., the PROG_B pin is permanently set to '0'). Notably, the EPA logic is lightweight as it occupies only 57 slices.

As shown in Figure A.1, the logic of the PicoBlaze (i.e., KCPSM3) and its attached EPA is triplicated to gain more reliability (i.e., TMR is used). Indeed, TMR is traditionally used in other reliability-oriented processors, such as LEON3-FT [Aeroflex Gaisler Inc., 2012]. Unlike these processors, when a voter detects a failure in any of the three cores of PicoBlaze, a scrubbing request for that specific core and its associated EPA logic is sent to the R3TOS configuration manager. In short, we have implemented a Fault-Tolerant PicoBlaze (called KCPSM3-FT) [Hong et al., 2012] that saves FPGA resources compared with related approaches that usually include redundant program memories (e.g., [Heiner et al., 2008]). The amount of saved resources increase thus with complex applications that require more program memory. In terms of logic, the KCPSM3-FT requires 558 slices in a Virtex-4 FPGA, from which the added logic (i.e., EPA) consumes only 171 slices (30%).

![Figure A.1: KCPSM3-FT: block diagram and interconnection in R3TOS HWuK](image-url)

In order to ease the development of applications using KCPSM3-FT, we have implemented a software tool that works in conjunction with standard tools provided by Xilinx Inc., i.e., KCPSM3 assembler and data2MEM. When used together, these tools allow for initialising an ECC-protected BRAM with an assembled program code.

In this thesis, the KCPSM3-FT has been used to implement both HWuK task scheduler and task allocator, where two complicated algorithms (i.e., FAEDF and EAC-Snake) that use most of PicoBlaze instructions are executed. We have checked that KCPSM3-FT has the ex-
act functionality as a standard PicoBlaze, with a speed loss less than 10% due to the synchronisation operations required to run from an ECC-protected BRAM. On the other hand, we have tested the improved fault-tolerance of KCPSM3-FT compared to a standard PicoBlaze by injecting faults both in their program BRAM as well as in their logic. However, as for R3TOS in general, we note that it should be convenient to carry out a more exhaustive dependability assessment of KCPSM3-FT (e.g., using a cyclotron).

While the idea of using ECC-protected BRAMs targeting a PicoBlaze was my own, I acknowledge that most of KCPSM3-FT implementation was done by Mr Chuan Hong. I designed its self-recovery mechanism and development framework.

This Appendix first explains the functioning of a standard Xilinx PicoBlaze processor and then switches to describe the implementation and functioning of KCPSM3-FT along with its development framework.

### A.1 Background: Standard PicoBlaze and ECC-Protected BRAMs

The PicoBlaze is an 8-bit soft-core microcontroller developed by Xilinx Inc. [Xilinx Inc., 2011b]. Its program memory is implemented using a 1024x18-bit BRAM, where each instruction is 18-bit wide and can be fetched in a single clock cycle. Overall, a PicoBlaze instruction cycle, i.e., an instruction fetch-decode-execute, is completed within 2 clock cycles (see Figure A.2).

As explained in chapter 2, an ECC-protected BRAM is a combination of two BRAMs, upper and bottom, and a built-in ECC coder-decoder, which implement the (72,64) Hamming code (see Figure A.3). Internally, the coder is responsible for generating 8 bits of ECC parity to be appended to the 64 data bits in every write operation, and the decoder processes the 72 bits read from the BRAM, i.e., the 64 data bits and the 8 parity bits which were generated by the coder. If there is no error, the original data is output. If there is a single-bit error, the
error is automatically corrected and output. Finally, if there is a multiple-bit error, the error cannot be corrected, but it is diagnosed through the status output, as shown in Table A.1.

Since one clock cycle is needed for decoding the ECC values, an ECC-protected BRAM data read has 2 clock cycles latency, i.e., 2 clock cycles are needed to fetch instructions from an ECC-protected BRAM. Nonetheless, as the ECC-protected BRAM is 64-bit data wide, up to three 18-bit PicoBlaze instructions can be accommodated in a single memory position. As illustrated in Figure A.4, in our KCPSM3-FT implementation, we have packed two instructions (lower and higher) in each memory position, thus allowing for exactly the same number of instructions as when using a standard PicoBlaze. Besides, we note that packing two instructions in a memory position is enough to hide the increased latency of the ECC-protected BRAM.

### A.2 The ECC Processor Adapter (EPA)

The EPA implements the interface between the PicoBlaze and the ECC-protected BRAMs (see Figure A.6). As two 18-bit instructions are packed in a 64-bit ECC-protected BRAM position, the EPA pre-fetches instructions from the latter memory in pairs. Note that this can
be compared to an instruction pipeline where most likely the immediate adjacent instruction is the instruction to be next executed. Hence, the least significant bit of the instruction address issued by the PicoBlaze, $\text{Addr}(0)$, is used to select between the two pre-fetched instructions that lie in the ECC-protected BRAM position with address $\text{Addr}(9..1)$ (i.e., a '0' selects the lower instruction and a '1' selects the higher instruction). When pre-fetching the instructions, 2 clock cycles are needed for reading from the ECC-protected BRAM and an extra clock cycle is needed for EPA decoding. Put together, the instructions are fetched with 3 clock cycles latency, i.e., the address is 3 clock cycles ahead of the instruction (see Figure A.5). Note that this delay is significant only in the first fetching cycle. Afterwards, “pipelining” allows for delivering a new instruction to PicoBlaze cores of the KCPSM3-FT every two clock cycles, as in the case with standard PicoBlaze, thus disguising the increased latency of ECC-protected BRAMs. In order to overcome the synchronisation problem due to the initial delay, a dummy instruction (NOP) is inserted to fill the time until the first two instructions are fetched from the ECC-protected BRAM at address 0x000. We use $\text{LOAD s0,s0}$ as NOP operation, which re-assings the content of register $s0$ to itself (i.e., 0x01000).
Figure A.6: KCPSM3-FT implementation
However, the synchronisation between the ECC-protected BRAM and the PicoBlaze will be lost again when executing certain flow control operations which interrupt the instruction pipeline, including jumps, branches, procedure calls, and procedure returns. This is why the instruction bus is an input to the control logic of the EPA, which also receives the processor’s Zero (Z) and Carry (C) flags to detect conditional flow branches\(^1\), as well as the interrupt signal to detect jumps to the Interrupt Service Routine (ISR) (i.e., address 0x3FF). The following sections describe the way that the EPA deals with these special situations where the next instruction fetch must be delayed.

The EPA also plays an important role in the reliability of KCPSM3-FT. Indeed, although an ECC-protected BRAM is able to present corrected data at its output when a single-bit error has occurred, the stored information in the memory array is not automatically fixed. The EPA includes thus a scrubbing logic which corrects errors in the program memory without requiring any intervention from the processor. This logic is coupled to the write port of the BRAM and receives as inputs the (corrected) data read as well as the status output (see Figure A.1). As shown in Figure A.7, the writing of the corrected data is delayed until PicoBlaze turns to fetch instructions from a different memory position, i.e., otherwise a collision would occur when accessing the same address in the BRAM from its two ports. When a double or an undetermined error is detected (i.e., \(\text{status}[1] = 1\)’), then the program code is corrupted and a full FPGA reconfiguration is forced via PROG_B pin. As explained in chapter 4 and 6, there are other errors in the HWuK that also require an FPGA reconfiguration to be corrected, being all of them combined using a NOR logic gate (see Figure A.1).

### A.2.1 JUMP Instruction

When a **JUMP** instruction is executed, the program counter of the PicoBlaze will point to the memory address specified in the instruction in the next cycle. As during startup, a dummy instruction (**NOP**) is inserted to deal with the increased latency of the ECC-protected BRAM (see Figure A.8). As an overhead, the inserted **NOP** instruction results in a 2 clock cycle loss any time a **JUMP** instruction is executed.

---

\(^1\)Despite the fact Zero and Carry flags are not standard ports in the PicoBlaze distribution (i.e., KCPSM3), they are easily accessible in the VHDL description. This is the only modification that needs to be done to the regular PicoBlaze, VHDL code and ports redefinition.
In order to handle conditional jump instructions, the Z and C flags are extracted from the PicoBlaze to the EPA. The latter checks whether the jump conditions are met, in which case a \textit{NOP} operation needs to be inserted. The conditional jump includes: \texttt{JUMP C} (Jump only if C flag = '1'), \texttt{JUMP NC} (Jump only if C flag = '0'), \texttt{JUMP Z} (Jump only if Z flag = '1'), and \texttt{JUMP NZ} (Jump only if Z flag = '0').

### A.2.2 \texttt{CALL} and \texttt{RETURN} Instructions

When a \texttt{CALL} instruction is executed, the subsequent program address is pushed into the stack of the PicoBlaze, and its program counter will point to the starting address of the invoked subprogram in the next cycle. This is the same as in the \texttt{JUMP} instruction and thus, a \texttt{NOP} operation is inserted to deal with the increased latency of the ECC-protected BRAM.

---

**Figure A.7:** Scrubbing a single error in the program memory

**Figure A.8:** \texttt{JUMP} instruction timing
At the end of the subprogram, a `RETURN` instruction is to be executed to return back to the address from which the subprogram was called, i.e., the instruction address pushed into the stack. For instance, in Figure A.9, when the `CALL` instruction is executed, the subsequent address (0x001) is pushed into the stack. This instruction is later popped out when the `RETURN` instruction is executed. However, as the address is one more cycle ahead of the instruction when using an ECC-protected BRAM than when using a standard BRAM, the pushed address is incremented by 2 instead of 1, resulting in a wrong return address. For instance, in Figure A.10a, 0x002 address is pushed into the stack instead of 0x001. To solve this, the EPA logic decrements the popped out address by 1 when a `RETURN` instruction is detected, as depicted in Figure A.10b. Since the `RETURN` operation directly manipulates the instruction address, another `NOP` operation has to be inserted to deal with the increased latency of the ECC-protected BRAM.

Finally, note that the Z and C flags are used to handle conditional call and return instructions; i.e., `CALL Z`, `CALL NZ`, `CALL C`, `CALL NC`, `RETURN Z`, `RETURN NZ`, `RETURN C` and `RETURN NC`.

### A.2.3 INTERRUPT and RETURNI Instructions

When an interrupt is received, the PicoBlaze finishes executing the current instruction, pushes the next instruction address into the stack, i.e., this address is to be popped out when returning from the ISR, and jumps to address 0x3FF, where the interrupt vector is located (see Figure A.11).

However, as the address is one more cycle ahead of the instruction when using an ECC-protected BRAM than when using a standard BRAM, the EPA must preempt the instruction immediately before the instruction preempted by the PicoBlaze, which has yet to be executed. This is shown in Figure A.12a. When the interrupt is received, the PicoBlaze preempts
Appendix A - Design and Implementation of a Fault-Tolerant PicoBlaze: KCPSM3-FT

(a) CALL

(b) RETURN

Figure A.10: CALL and RETURN instructions timing in KCPSM3-FT

Figure A.11: INTERRUPT and RETURNI operation timing in PicoBlaze

the instruction at address 0x008 (ADD s0, s1) and the EPA preempts the instruction at address 0x007 (INPUT s0, s1). Note that the EPA includes a specific register to temporarily save the latter instruction (see Figure A.6).

In the scope of KCPSM3-FT, RETURNI (return from an ISR) differs from RETURN (return from a subprogram) as the instruction that is missed when executing the former is not necessarily the one stored in the previous address. Therefore, instead of returning to the
previous instruction address, as explained in section A.2.2, the EPA inserts the instruction that was saved when the interruption was received. This is depicted in Figure A.12b. When returning from the ISR, the instruction preempted by the EPA (INPUT s0,01) is inserted while fetching the instruction preempted by the PicoBlaze (ADD s0,s1). This allows for disguising the increased latency of the ECC-protected BRAM.

![Figure A.12: INTERRUPT and RETURNI operation timing in KCPSM3-FT](image)

A.3 Development Framework

The development framework covers all of the tools that need to be used to set-up a KCPSM3-FT to implement a specific functionality. It includes two standard tools provided by Xilinx Inc., i.e., KCPSM3 assembler and data2MEM, and a specifically designed script, which is programmed in the C language.
The HEX file generated by Xilinx KCPSM3 assembler is suitable to be used with a 1024x18 standard BRAM, but not with a 512x64 ECC-protected BRAM. Indeed, in the latter case, the ECC bits must be generated offline in order to be written together with the data information in the bitstream. In light of this, we have developed a script that processes the program code generated by the KCPSM3 assembler prior to using the data2MEM tool. Namely, our script packages two instructions in a single memory position and generates the corresponding ECC bits. Moreover, as data2MEM tool can only be used with RAMB18 (i.e., 1024x18) and RAMB36 (i.e., 512x36) memories, our script decomposes the program code and ECCs into two pieces to be loaded in the two BRAM36, upper and bottom, that are internally used in the ECC-protected BRAM (see Figure A.3). The script hence creates a pair of BMM² and MEM³ files that are inputs to the data2MEM tool. The latter is sequentially executed to update both RAMB36s; i.e., the BIT file obtained when executing data2MEM for the first time becomes the input to the next execution. This design flow is depicted in Figure A.13.

---

²A BMM file defines the exact placement and features of the BRAM memory where to load the program executable code.
³A MEM file is essentially a HEX file (output from KCPSM assembler) with '@0' as the first line.
Bibliography


360


361


362


