Simulation Models of Shared-Memory Multiprocessor Systems

Paul Coe

Thesis presented for the degree of
DOCTOR OF PHILOSOPHY

University Of Edinburgh
February 2000
Abstract

Multiprocessors have often been thought of as the solution to today's ever increasing computing needs; but they are expensive, complex and difficult to design. This thesis focusses on the development of multiprocessor simulations that would aid the design and evaluation of such systems.

The thesis starts by outlining the various possibilities for multiprocessor design and discusses some of the more common problems that must be addressed. A selection of simulation environments and models that have been developed to study complex computer systems are then described.

The major problem with these simulation systems is that they generally focus on a small area of multiprocessor systems design in order to produce fast simulations that generate results quickly; consequently they provide very little flexibility and room for exploration. The aim of this project was to design and implement a flexible multiprocessor model within the HASE simulation environment, enabling the designer to explore a large design space with a minimum of effort, focussing more on flexibility and less on simulation speed. A parameterised simulation model has been developed that presents the designer with many design options with which to experiment. The parameters allow simple alternatives to be explored, for example, different component speeds or bus widths, as well as more complicated features, for example, coherence protocols, synchronisation primitives and architecture configurations. The model was designed in a modular manner that allows new parameter values to be incorporated, as well as new implementations of the various entities. To support this new model, the HASE system was extended to provide better support for multiprocessor modelling.

A selection of experiments was performed using the model and simulation framework. These experiments not only illustrate the capabilities of this model, but also demonstrate the importance of simulation as a tool for studying multiprocessor designs.
Acknowledgements

I would like to thank my supervisor, Professor Roland Ibbett, for his invaluable advice, support and encouragement during my time in the department.

Many thanks must go to Lawrence Williams and Adam Donlin, with whom I shared an office, for their friendship, help and advice during my Ph.D. Thanks also to other friends in the department, especially Christos, Dom, Jon, Sal, and Bob, who made my time most enjoyable.

My family, especially my Mum and Dad, must be thanked for their support and belief in me throughout my University life. My partner Kirsti also deserves special thanks, for everything, as always.
Declaration

I declare that this doctoral thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text. The following articles were published during my period of research. Certain material and concepts from these publications will necessarily be presented within the body of this work.


Paul Coe
14th December 1999
Table of Contents

ABSTRACT .......................................................................................................................... 1

ACKNOWLEDGEMENTS .................................................................................................... 2

DECLARATION ..................................................................................................................... 3

TABLE OF CONTENTS ........................................................................................................ 4

LIST OF FIGURES .............................................................................................................. 10

LIST OF TABLES .................................................................................................................. 11

LIST OF GRAPHS ................................................................................................................ 12

INTRODUCTION .................................................................................................................... 13

1.1 PREAMBLE .................................................................................................................. 13
1.2 AIMS ............................................................................................................................ 16
1.3 THESIS OUTLINE ....................................................................................................... 17

BACKGROUND .................................................................................................................... 20

2.1 MULTIPROCESSOR ARCHITECTURES ........................................................................... 20

2.1.1 Message Passing Multiprocessors ........................................................................... 21
2.1.2 Shared-Memory Multiprocessors .......................................................................... 23
2.1.3 Hybrid Architectures ............................................................................................. 30

2.2 CACHE COHERENCE ................................................................................................... 32

2.2.1 The Cache Coherence Problem .............................................................................. 32
2.2.2 No Local Caches .................................................................................................... 34
2.2.3 Broadcast Coherence Protocols ............................................................................. 35
2.2.4 Directory Coherence Protocols ............................................................................. 36
2.2.5 Software Approaches ............................................................................................. 38

2.3 SYNCHRONISATION ................................................................................................... 40

2.3.1 Spin-locks ............................................................................................................... 40
2.3.2 Barriers .......................................................... 41

2.4 MODELS OF MEMORY CONSISTENCY ... 42

2.4.1 Sequential Consistency ...................... 42

2.4.2 Weak Ordering .................................. 43

2.4.3 Release Consistency ......................... 44

MULTIPROCESSOR MODELLING AND SIMULATION ............................. 46

3.1 EVALUATION USING MULTIPROCESSOR SYSTEMS .................... 46

3.2 MATHEMATICAL AND ANALYTICAL MODELS ......................... 48

3.3 SIMULATION OF MULTIPROCESSOR SYSTEMS ...................... 51

3.3.1 General Simulation Environments .............. 51

3.3.2 Multiprocessor Simulators .................... 54

HIERARCHICAL COMPUTER ARCHITECTURE DESIGN AND SIMULATION ENVIRONMENT ............................................. 71

4.1 THE EVOLUTION OF HASE .................................. 71

4.2 RECENT DEVELOPMENTS IN HASE .................... 73

4.2.1 Project Data Storage ......................... 73

4.2.2 Discrete Event Simulation Engine .............. 76

4.2.3 Modal Operation ................................ 77

4.2.4 Model Validation and Library Management Tool .................. 79

4.2.5 Microsoft Windows Version of HASE ............... 79

4.3 THE CURRENT HASE SYSTEM ..................... 80

4.3.1 Internal Architecture Representation .......... 82

4.3.2 Model Hierarchy ................................ 84

4.3.3 Architecture Templates ....................... 87

4.3.4 Visualisation .................................... 88

4.3.5 Model Experimentation ....................... 92

DESIGN OF THE MULTIPROCESSOR MODEL .................................... 93

5.1 MOTIVATION ........................................... 93

5.2 DESIGN OF THE MULTIPROCESSOR ARCHITECTURE MODEL .......... 96

5.2.1 Processor Entity ................................ 98

5.2.2 Cache Entity .................................. 105
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.3</td>
<td>Memory Entity</td>
<td>109</td>
</tr>
<tr>
<td>5.2.4</td>
<td>The Basic System</td>
<td>112</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Bus Entity</td>
<td>113</td>
</tr>
<tr>
<td>5.2.6</td>
<td>A Shared-Memory Multiprocessor System</td>
<td>116</td>
</tr>
<tr>
<td>5.2.7</td>
<td>Multiple Common Memory Entities</td>
<td>118</td>
</tr>
<tr>
<td>5.2.8</td>
<td>Distributed Shared-Memory Multiprocessor</td>
<td>121</td>
</tr>
<tr>
<td>5.2.9</td>
<td>Network Interface Entity</td>
<td>124</td>
</tr>
<tr>
<td>5.2.10</td>
<td>Multiprocessor Model Design Summary</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>IMPLEMENTATION OF THE MULTIPROCESSOR MODEL</td>
<td>128</td>
</tr>
<tr>
<td>6.1</td>
<td>ENTITY COMMUNICATION</td>
<td>129</td>
</tr>
<tr>
<td>6.2</td>
<td>PROCESSOR ENTITY</td>
<td>134</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Data Placement Policy</td>
<td>136</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Synchronising the Application with the Simulation</td>
<td>139</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Implementation of the Synchronisation Primitives</td>
<td>140</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Other Implementation Issues of the Processor Entity</td>
<td>145</td>
</tr>
<tr>
<td>6.3</td>
<td>CACHE ENTITY</td>
<td>146</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Cache Coherence Protocols</td>
<td>147</td>
</tr>
<tr>
<td>6.4</td>
<td>MEMORY ENTITY</td>
<td>161</td>
</tr>
<tr>
<td>6.5</td>
<td>BUS ENTITY</td>
<td>163</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Extending the Bus Entity for Distributed Shared-Memory Multiprocessors</td>
<td>166</td>
</tr>
<tr>
<td>6.6</td>
<td>NETWORK INTERFACE</td>
<td>168</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Upper Level Coherence Protocols</td>
<td>171</td>
</tr>
<tr>
<td>6.7</td>
<td>COMPLETE SYSTEMS</td>
<td>176</td>
</tr>
<tr>
<td></td>
<td>HASE EXTENSIONS</td>
<td>183</td>
</tr>
<tr>
<td>7.1</td>
<td>PARAMETER DEPENDENT ARRAYS</td>
<td>183</td>
</tr>
<tr>
<td>7.2</td>
<td>LINKER PARAMETERS</td>
<td>184</td>
</tr>
<tr>
<td>7.3</td>
<td>NEW TEMPLATES</td>
<td>188</td>
</tr>
<tr>
<td>7.4</td>
<td>EXPERIMENT CONTROL MECHANISM</td>
<td>189</td>
</tr>
<tr>
<td>7.5</td>
<td>OTHER EXTENSIONS</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td>SIMULATION EXPERIMENTS</td>
<td>193</td>
</tr>
</tbody>
</table>
C.2 THE BDIV FUNCTION
C.2.1 C Code for bdiv
C.2.2 Assembly Code for bdiv
C.2.3 Extended C Code for bdiv

C.3 THE BMODD FUNCTION
C.3.1 C Code for bmodd
C.3.2 Assembly Code for bmodd
C.3.3 Extended C Code for bmodd

C.4 THE BMOD FUNCTION
C.4.1 C Code for bmod
C.4.2 Assembly Code for bmod
C.4.3 Extended C Code for bmod

C.5 THE DAXPY FUNCTION
C.5.1 C Code for daxpy
C.5.2 Assembly Code for daxpy
C.5.3 Extended C Code for daxpy

C.6 THE BLOCKOWNER FUNCTION
C.6.1 C Code for blockowner
C.6.2 Assembly Code for blockowner
C.6.3 Extended C Code for blockowner

C.7 THE LU FUNCTION
C.7.1 C Code for lu
C.7.2 Assembly Code for lu
C.7.3 Extended C Code for lu

C.8 THE SLAVESTART FUNCTION
C.8.1 C Code for SlaveStart
C.8.2 Assembly Code for SlaveStart
C.8.3 Extended C Code for SlaveStart

APPENDIX D

D.1 CACHE ASSOCIATIVITY AND BLOCK REPLACEMENT POLICY
EXPERIMENT
D.1.1 Parameter Values
D.1.2 Table of Results ................................................................. 276

D.2 WRITE POLICY AND ALLOCATION POLICY EXPERIMENT .......... 277

D.2.1 Parameter Values ............................................................ 277

D.2.2 Table of Results .............................................................. 278

D.3 MULTIPLE LEVELS OF CACHE EXPERIMENT .......................... 279

D.3.1 Parameter Values ............................................................ 279

D.3.2 Table of Results .............................................................. 280

D.4 CACHE COHERENCE PROTOCOL EXPERIMENT .................... 281

D.4.1 Parameter Values ............................................................ 281

D.4.2 Table of Results .............................................................. 282

D.5 NUMBER OF PROCESSORS IN A BUS-BASED MULTIPROCESSOR EXPERIMENT ................................................................. 283

D.5.1 Parameter Values ............................................................ 283

D.5.2 Table of Results .............................................................. 284

D.6 DIFFERENT PROCESSOR SPEEDS EXPERIMENT .................... 285

D.6.1 Parameter Values ............................................................ 285

D.6.2 Table of Results .............................................................. 286

D.7 SYNCHRONISATION PRIMITIVE IMPLEMENTATION EXPERIMENT 287

D.7.1 Parameter Values ............................................................ 287

D.7.2 Table of Results .............................................................. 288

D.8 CACHE SIZE IN A CLUSTERED SYSTEM ............................... 289

D.8.1 Parameter Values ............................................................ 289

D.8.2 Table of Result ............................................................... 290

D.9 CLUSTER CONFIGURATION EXPERIMENT ............................. 291

D.9.1 Parameter Values ............................................................ 291

D.9.2 Table of Results .............................................................. 293
## List of Figures

1.1 HASE main display ................................................................. 18
2.1 A distributed memory message passing multiprocessor .................. 22
2.2 A centralised shared-memory multiprocessor ................................ 24
2.3 A COMA shared-memory multiprocessor .................................... 27
2.4 A clustered NUMA shared-memory multiprocessor ....................... 29
2.5 Code to illustrate cache coherence ............................................ 33
2.6 Illustration of cache coherence ................................................ 33
4.1 An example EDL file ............................................................... 75
4.2 An example EL file ............................................................... 75
4.3 HASE main menu bar in Design and Simulate System modes of operation ... 78
4.4 A memory entity pull-down menu in Design and Simulate System modes ... 78
4.5 HASE software architecture ................................................... 81
4.6 A HASE memory entity .......................................................... 83
4.7 Sample HASE entity parameters ............................................. 84
4.8 A multiprocessor shown at a high level ..................................... 85
4.9 A multiprocessor shown at a lower level ................................... 86
4.10 Simulation hierarchy viewer ................................................... 87
4.11 HASE animator control panel ................................................ 89
4.12 Communication protocol viewer .............................................. 90
4.13 Timing diagram ................................................................. 91
4.14 Timing data displayed as percentages ...................................... 91
4.15 Experiment control panel ...................................................... 92
5.1 A clustered distributed shared-memory multiprocessor .................. 98
5.2 Three possible processor configurations .................................... 99
5.3 The structure of a cache line .................................................. 105
5.4 A uniprocessor system .......................................................... 112
5.5 A shared-memory multiprocessor system ................................... 117
5.6 Multiple common memory multiprocessor .................................. 119
5.7 A distributed shared-memory multiprocessor ............................... 121
5.8 A clustered distributed shared-memory multiprocessor ................................ 123
5.9 Position of the network interface entity .................................................. 124
6.1 Example EDL description of ports and links ............................................ 130
6.2 Representation of the EDL description of Figure 6.1 .................................. 130
6.3 The message structure used for requests .................................................. 132
6.4 The message structures used for results and acknowledgements .............. 133
6.5 Sample EDL for link parameter definition and use .................................... 133
6.6 Placement of $lu$ data structures in a single memory entity ....................... 137
6.7 Possible data placement policies ............................................................... 138
6.8 Steps involved in the conversion of the daxpy function ............................. 141
6.9 Ticket lock ............................................................................................... 142
6.10 Sense-reversing centralised barrier ......................................................... 144
6.11 The network interface entities and their communication links .................. 169
6.12 The structure of a node ........................................................................... 172
6.13 HASE representation of a single processor system ................................... 177
6.14 HASE representation of a four-node bus-based multiprocessor ............... 179
6.15 Multiple memory multiprocessor ............................................................. 181
6.16 Clustered distributed shared-memory multiprocessor using a crossbar
   network ....................................................................................................... 182
7.1 Generation of simulation executable from the EDL and hase files ............... 186
7.2 EDL description of a bus-based multiprocessor entity ................................ 189
8.1 Single processor system ............................................................................ 195
8.2 An eight-node bus-based multiprocessor system ...................................... 205
8.3 A typical clustered distributed shared-memory multiprocessor .................. 218

**List of Tables**

5.1 Parameters of the processor entity ......................................................... 104
5.2 Parameters of the cache entity ............................................................... 110
5.3 Parameters of the memory entity ............................................................ 112
5.4 Parameters of the bus entity ................................................................. 115
5.5 Parameters of the multiprocessor entity ............................................................. 118
5.6 Parameters of the multiple common memory multiprocessor entity .......... 120
5.7 Parameters of the distributed shared-memory multiprocessor entity ......... 122
5.8 Parameters of the network interface entity .................................................... 126
8.1 Cache hit rates for different associativities and replacement policies ......... 198

List of Graphs

8.1 System performance using different cache associativities for different
    replacement policies ....................................................................................... 196
8.2 System performance using different write and allocation policies for various
    cache sizes .................................................................................................... 200
8.3 System performance for different configurations of level 2 cache ............ 203
8.4 System performance for different coherence protocols and cache sizes ...... 206
8.5 Multiprocessor performance for different numbers of processors ............ 209
8.6 The effects of different processor speeds on performance ....................... 213
8.7 Simulated times and simulation execution times for different implementations
    of the synchronisation primitives ................................................................ 216
8.8 Performance of a 4 cluster system (4 processors per cluster) with different
    cache sizes .................................................................................................... 219
8.9 Performance of different cluster configurations for different interconnection
    network speeds ........................................................................................... 221
Chapter 1

Introduction

1.1 Preamble

In today’s technological world a huge quantity of data is continually being generated and analysed, more and more complex simulations of physical and environmental systems are being developed and larger, more powerful file servers are required to support all of this. The rapid rate at which processors have improved over the years to try to keep pace is quite remarkable; however each improvement moves the chips closer and closer to the physical limits of the materials used. It is doubtful that processor technology can continue to develop at its current rate for too much longer, so alternative solutions have to be explored to match the demand for more processing power.

An approach that has been studied in detail since the 1960’s is to include multiple processors within a system, rather than trying to increase the performance of a single processor. Multiprocessor machines have often been thought of as the answer to the computing needs of the future. Construction of multiprocessors from off-the-shelf processors would enable competitive performance to be offered at a fraction of the cost of traditional mainframes. The perceived advantages gained by replicating parts
of a system include better performance (more processors are better than one), reliability (if one processor fails the system does not stop) and incremental upgrades (adding more of the replicated parts).

However, all is not as straightforward as it seems. By including more processors the system as a whole becomes much more complex, resulting in longer design times. This problem becomes even worse when the relatively short design times for uniprocessors are considered. If care were not taken, the multiprocessor system based on older uniprocessor technology would not provide enough of a performance advantage over the latest uniprocessor systems to justify the extra expense. Not only have multiprocessors been expensive, but there has also been a lack of programming support, in particular good parallelising compilers, debuggers and performance analysis tools, requiring programmers with expert knowledge to fine tune applications for a particular machine.

In theory, connecting together multiple processors could offer a performance that is equal to the sum of the performance of the individual processors. However, this “Holy Grail” of multiprocessor performance is impossible to achieve, due to many factors (including non-parallel portions of code, and delays introduced through communication and synchronisation). The goal of the designer is therefore to produce a system that is as close to the ideal as possible, so comprehensive design tools are required to enable the designer to better explore this design space.

A variety of systems have been designed in the past four decades to try to achieve optimum performance (this area is looked at in more detail in Chapter 2). More recently, shared-memory multiprocessors have emerged as a popular choice in the ongoing quest for better performance. Shared-memory machines provide a simpler programming interface, as all processors have access to all memory locations without the need for complex communication with another processor, allowing code and data to be shared efficiently. This also enables data structures from sequential code to be retained; simply adding synchronisation may be enough in some applications to achieve correct operation of the code in parallel.
However, the programming advantages of using shared-memory are offset by the problem of a more complex system design, which has historically limited their scalability. Chapter 2 also discusses in more detail the different types of multiprocessor architectures that have been proposed and the programming models that they support.

There is a myriad of design decisions and trade-offs that must be made when creating a multiprocessor system, including how many processors to use, how they should be connected together, what memory configuration should be used and how the processors will communicate. Many of these decisions will have a significant effect on the final system performance and, although the designers' experience can aid in this decision making, it is impossible to predict the effect of all of these decisions on the overall performance.

Complicating the design process is the fact that one part of the system may have influence over many other parts; consequently evaluating the effects of individual decisions in isolation will not always give a true indication of the final performance. Many techniques have been developed to aid a designer in predicting the performance of a system, ranging from analytical techniques for evaluating individual parts (for example the interconnection network used), to a complete system simulation. A range of the various techniques used will be discussed in depth in Chapter 3.

The complex interactions of the different components of a multiprocessor system require suitable design tools to be created to enable design trade-offs to be explored and evaluated with a minimum of effort. The designer needs to be able to assess the impact of, for example, adding more processors, adding another level of cache memory or changing the method of synchronisation used without rewriting the simulation or redesigning the system to incorporate the suggested changes.
The vastness of the design space of a multiprocessor system highlights the need for an environment that will allow the designer to explore portions of this space rapidly and with ease. As well as being able to examine the impact of a specific design change, the environment also needs to cater for more complex demands, for example, evaluating different combinations of a range of implementation choices for a range of system components.

Much work has been carried out in providing simulations of specific multiprocessor systems that provide a platform to assess the performance of new hardware or software techniques, parallel applications or benchmarks. This work has predominantly been aimed at either evaluating new techniques and comparing them to existing techniques for a specific aspect of a whole system, or in providing a fast simulation of a specific multiprocessor to enable complete operating system and application code to be executed. Very little work has been reported on the development of a parameterised model of a parallel architecture that would enable designers to easily explore and evaluate different design trade-offs. There is also a need for an environment that allows the designer to take advantage of the parameters offered by the architecture model, enabling multiple experiments to be performed which compare many different combinations of system components with a minimum of effort.

1.2 Aims

The main aim of the work described here was to attempt to address the problem of a lack of tools to explore the multiprocessor design space. Two main areas were identified that had to be addressed to solve this problem. The first area identified was the need to develop a model of a multiprocessor system that had enough parameters to enable a wide range of design alternatives to be evaluated. The model developed should be easily extendable, to adapt to new ideas and techniques or to allow different styles of multiprocessor architecture to be considered. The model should include parameters to enable a designer to explore the effects of different
numbers of processors, the grouping and methods of interconnection of the processors, the methods of maintenance of system coherence and different configurations of the cache and memory components.

The second area identified was the development of an environment that allows the designer to take advantage of this parameterised multiprocessor model, enabling easy and comprehensive exploration of different architecture configurations and parameter settings.

The final specification of the work performed was that it be designed to be compatible with HASE (Hierarchical computer Architecture design and Simulation Environment) that was already under development at the University of Edinburgh. HASE is a hierarchical, object-oriented environment that allows architectures to be displayed graphically and simulations of their behaviour to be created. The results of a simulation can be viewed in a variety of ways, including animation of the architecture display and logic analyser style timing diagrams. Figure 1.1 shows the HASE display containing a representation of a multiprocessor system that contains two clusters each with four processors, four caches, a bus, memory and a network interface.

1.3 Thesis Outline

This thesis is organised in the manner outlined below:

Chapter 2 presents background information on different styles of multiprocessor architecture. The cache coherence problem is discussed together with a range of the solutions that have been proposed. The chapter also describes synchronisation and memory consistency models and how they affect the system.
Chapter 3 presents previous approaches that have been used to model multiprocessor architectures. Analytical models are discussed, along with simulation models and simulation environments.

Chapter 4 discusses HASE, the simulation environment used for the work carried out. The developmental history of HASE is detailed, along with an illustration of the main features of the environment.
Chapter 5 describes in depth the design of the parameterised multiprocessor model created. The design of each of the entities of the model is discussed, along with the parameters that each entity supports and how they affect the architecture.

Chapter 6 describes the implementation of the multiprocessor model and the implementation of the parameter options it supports. It describes in detail the implementation of the individual entities and how they are combined to construct complete systems.

Chapter 7 describes the extensions made to the HASE environment that allow a designer to take advantage of architecture parameters of the model created.

Chapter 8 presents the results obtained by running code from the SPLASH-2 [Woo95] benchmark suite through the multiprocessor model created, allowing different architectures and architectural features to be explored in detail. Results are presented to illustrate how different architecture abstractions can change simulation execution time, and impose limits on accuracy and/or parameter options. The significance of the results obtained from the simulations is also analysed and different architecture configurations are discussed in terms of the benchmark code executed.

Chapter 9 discusses the advantages of using HASE, when combined with the parameterised shared-memory model, for exploration and evaluation of different distributed shared-memory multiprocessors. Future extensions to HASE and the multiprocessor model that would enable a wider range of systems to be represented are also discussed.
Chapter 2

Background

This chapter describes the terms, concepts and problems (along with a selection of the proposed solutions) of multiprocessor systems. It begins by presenting an overview of the different types of multiprocessor system and their advantages and disadvantages. Specific areas of shared-memory multiprocessor systems design are then described, including cache coherence, memory consistency models and synchronisation.

2.1 Multiprocessor Architectures

In 1972 Flynn proposed a simple model for categorising multiprocessor systems [Fly72]. The categories proposed were based on the parallelism in the instruction and data streams. The four categories were:

1. **SISD** (Single Instruction stream, Single Data stream) – this is a uniprocessor architecture.
2. **SIMD** (Single Instruction stream, Multiple Data stream) – in this architecture each processor executes the same instruction, but on a different data set.
3. **MISD** (Multiple Instruction stream, Single Data stream) – no commercial machine has currently been constructed of this type.

4. **MIMD** (Multiple Instruction stream, Multiple Data stream) – in this architecture each processor executes its own sequence of instructions on its own data set.

The SIMD class of machine was originally the popular choice for multiprocessor systems, as it dealt well with the parallelism of arrays in loops and applications with a high degree of data parallelism. However, SIMD is no longer the favoured class of architecture for general-purpose multiprocessor systems, mainly due to the lack of flexibility of SIMD machines, i.e., there is a large class of problems which cannot be tackled efficiently using this style of machine. Also, SIMD machines cannot take advantage of performance and cost by using state of the art processor technology, as the processor used must be custom designed and not off-the-shelf.

MIMD has emerged as the class of machine for general-purpose multiprocessor systems. Unlike SIMD, MIMD systems can use off-the-shelf processors and therefore benefit from all the advantages of using up-to-date technology. MIMD machines are also very flexible; they are able to function as a single-user machine running one application, as a multi-user machine running many tasks simultaneously, or as a combination of the two. The communication model used, either message passing or shared-memory, is commonly used to classify MIMD multiprocessors; both models will be discussed in the following sections. Work has been carried out in the literature, for example, by Klaiber and Levy [Kla94], to compare the different communication models.

### 2.1.1 Message Passing Multiprocessors

Message passing multiprocessors usually consist of a set of processing nodes connected together by an interconnection network. Main memory is distributed amongst the nodes and the processors communicate and share data by sending and receiving messages. Figure 2.1 shows the typical structure of a distributed memory, message passing multiprocessor.
By distributing the memory amongst the processing nodes, message passing multiprocessors can achieve good levels of scalability. Providing local memory that is only accessible by the local processor allows the processor to perform most of its accesses to local memory locations with a relatively low latency. This is possible as the processor does not require interaction with any of the other nodes, and requests are not required to traverse the interconnection network. Off-the-shelf processors can be used to construct this type of multiprocessor, enabling large configurations to be constructed at a reasonable cost.

The communication protocols enforced by message passing machines have become standardised through the use of libraries such as MPI (Message Passing Interface) and PVM (Parallel Virtual Machine). These libraries have been implemented on many of the common machines, making it easier to port code between different multiprocessor platforms (a task that would be much more difficult without a common set of communication primitives).

The advantages outlined above have proved extremely tempting to industry, and message passing machines have proved very popular, with many research and commercial machines being constructed. Early message passing machines include the Cosmic Cube from Caltech [Sei85] and the Intel iPSC 860 [Bar91]. More recent
systems constructed to conform to this style of multiprocessor architecture include the Intel Paragon [Liu95] and the CM-5 from Thinking Machines [Kar95]. Now that workstations are becoming very powerful, techniques have also been developed to construct message passing multiprocessor systems from workstations connected together by a standard network (for example an Ethernet), which provides very inexpensive multiprocessor systems. These systems have become known as Networks of Workstations or NOW systems.

These advantages come at a price however. Communication between the processors is explicit, requiring programmers and compilers to pay considerable attention to communication and its costs. This complicates the programs created and makes writing them much more difficult.

The assignment of instructions and data between the processors is also extremely critical for obtaining good performance, because the migration of instructions and data structures between different processing nodes is a very costly operation. The assignment process is split into three problems, namely: partitioning, scheduling and data placement. Partitioning is the problem of splitting a program into a set of tasks, scheduling involves assigning tasks to processing nodes, and data placement deals with the assignment of data structures to memory modules. All three have a critical impact on the performance of parallel applications on message passing multiprocessors. For example, if a task is scheduled to execute on a processor and the data structures used by the task are placed in the local memory of a different node, large amounts of communication will result, either through the migration of the whole of the data to the other node, or through reading and writing all of the individual memory locations on request.

2.1.2 Shared-Memory Multiprocessors

Shared-memory multiprocessors use a centralised main memory and a single address space that allows all of the processors in the system to access any memory address. Communication between processors is provided through the shared addresses, i.e.,
one processor writes to an address that can then be read by another processor. Figure 2.2 shows the structure of a typical centralised shared-memory multiprocessor.

![Diagram of a centralised shared-memory multiprocessor](image)

**Figure 2.2:** A centralised shared-memory multiprocessor

By using a centralised shared-memory structure, many of the programming problems of message passing machines can be solved. The single address space makes programming simpler, as explicit send and receive calls are not required. Partitioning, scheduling and data placement are no longer critical problems as all processors have equal access to all memory locations.

The ease of programming comes at the expense of scalability. It is impossible to connect a large number of processors to a shared main memory and maintain a reasonable level of performance due to high levels of contention for the memory and interconnection network. Other problems, which are discussed later in this chapter, are also introduced by using a shared-memory multiprocessor system, for example, cache coherence.

The simple programming model, combined with good performance with small numbers of processors and low cost has meant that shared-memory multiprocessors have become popular for small-scale, relatively inexpensive multiprocessors solving small-scale problems. This style of machine became popular in the 1980's with the advent of the Synapse N+1 machine [Fra84]. Other machines quickly followed,
including the Sequent Symmetry [Lov88], the Dragon computer from the Xerox Corporation [McC85] and the Firefly multiprocessor workstation from the Digital Equipment Corporation [Tha87]. These machines are very similar in structure; they all use a shared bus to connect the processors and their caches to a common memory module. Their main differences are in how they maintain cache coherence (see Section 2.2).

The system outlined above is commonly referred to as a Uniform Memory Access shared-memory multiprocessor, as all processors take the same amount of time to access any location in main memory. Other forms of shared-memory multiprocessor have been proposed, for example, Non-Uniform Memory Access and Cache-Only Memory Access machines. These will be discussed below.

Non-Uniform Memory Access Machines

Non-uniform memory access (NUMA) shared-memory multiprocessors distribute the memory between the processing nodes, in a manner similar to message passing systems. The structure of a NUMA shared-memory multiprocessor is very similar to that of a message-passing multiprocessor (see Figure 2.1).

NUMA systems have been developed in an attempt to make shared-memory multiprocessors more scalable, without having to sacrifice the simpler programming model. By distributing the memory modules amongst the nodes, the machine becomes more scalable and more nodes can now be added without increasing the contention for a shared memory module. However, by adding memory to the nodes, the access times for different memory addresses can vary drastically. Access to the local memory is fast as it requires no network accesses. In contrast, accesses to memory addresses contained in remote memory modules require the request to traverse the network to access the remote memory; this process can take many more cycles than the local access. Remote accesses themselves may have variable access times, depending on which memory module is to be accessed.
The non-uniform nature of the accesses introduces problems of consistency, as memory operations may complete in a different order from that in which they were issued, causing problems with program correctness. This is discussed further in Section 2.4. There are also added coherency problems with distributing memory amongst the nodes, for example, requesting data from a memory location that has been modified by a third node requires requests to be forwarded to the third node. These problems will be discussed in Section 2.2.

The attractiveness of the shared-memory programming model, combined with the scalability of message passing machines has made this approach to multiprocessing popular. Early attempts at creating scalable shared-memory multiprocessors include the NYU Ultracomputer [Edl85], the IBM RP3 [Pfi85a], the University of Illinois Cedar Project [Gaj83] and the BBN Butterfly [BBN87]. Although they all implemented variations of the distributed, non-uniform memory access model, none provided hardware support for cache coherence. The techniques developed in these early systems have been carried through to some recent systems that also provide a non-coherent view of memory, including the Cray T3D [Kar95]. Another approach to scalable distributed shared-memory machines includes the MIT Alewife [Aga99] which connects up to 128 processors using a 2D mesh interconnect and maintains coherence using a directory protocol (see Section 2.2.4).

**Cache-Only Memory Access Machines**

Cache-only memory access (COMA) machines also distribute memory amongst the processing nodes. Figure 2.3 shows the typical structure of a COMA shared-memory multiprocessor architecture.

COMA architectures differ from NUMA architectures in that they allow data to migrate between the memory modules, so data does not reside in a specific memory module. In NUMA architectures all data have an associated home memory module to which all requests can be issued. However, in COMA architectures there is no notion of a home memory module, so a lookup must be performed to find the
memory module at which the data are currently residing. COMA architectures must also include measures to ensure that data replaced in a memory module is written back to disk if it is the last copy held in any of the memory modules. It is unnecessary to check if there are any other copies in a NUMA architecture because the data only resides in one memory module. The advantage of the COMA approach is that frequently used data originally belonging to a remote memory module can be migrated to the local memory module, reducing future access time.

![Figure 2.3: A COMA shared-memory multiprocessor](image)

Shared-memory multiprocessor systems based on the COMA style of architecture have been produced commercially and in research. This style of architecture has been less popular when building actual machines due to the more complicated memory modules and the need for lookup tables to find which memory module an item of data is currently held in. The most notable machine that fits into this class of architecture is the KSR-1 (Kendall Square Research) machine [Ram93].

Research has been performed to compare the NUMA and COMA styles of shared-memory architecture [Sin93]. Neither style appears to offer a significant performance advantage and it appears to be application dependent as to which one performs best. Singh et al [Sin93] found that COMA architectures favoured applications where capacity misses dominated (as the memory modules act like large
caches) whereas NUMA architectures favoured applications where coherence misses dominated.

**Clustered Multiprocessor Systems**

Distributed memory techniques have enabled shared-memory multiprocessors to be scaled to large numbers of processors. However this scalability comes at a cost. It no longer becomes feasible to use a bus to connect the processing nodes together as, apart from the electrical and physical constraints, connecting too many nodes would cause a high level of contention for the shared bus. Instead other forms of interconnect must be used, for example, meshes or multistage networks, as these allow more nodes to be connected together. The main disadvantage of these types of network is the latency, i.e., the time taken to send a message to another node. Unlike buses that broadcast to all nodes in a roughly constant time, other networks require messages to be routed through other nodes or intermediate switches before they arrive at their destination. This means that sharing data amongst processors that are linked using non-bus network structures can be a costly operation.

To overcome the limitations of using non-bus networks, a clustered approach to shared-memory multiprocessors has emerged. In this type of architecture the node of the network is no longer made up of a processor, its caches and a part of main memory; it now consists of several processors and their caches sharing a part of main memory. The processors are connected to local memory using a shared bus, as in small-scale shared-memory systems, and these multiprocessor nodes are then connected together using a separate interconnection network. Figure 2.4 shows the typical structure of a clustered NUMA shared-memory multiprocessor. A similar structure can be used for COMA styles of architecture.

This approach combines the advantages of using buses to connect together small numbers of processors that are sharing memory, with the scalability of distributed memory.
The clustered NUMA and COMA approaches to shared-memory multiprocessing have become known as Distributed Shared-Memory (DSM) systems. Clustering processors into shared-memory nodes has recently become the most popular method employed for constructing shared-memory multiprocessors. The Stanford DASH [Len92] was one of the first hierarchical machines to be constructed and consisted of four processors per cluster, each with local caches, connected to memory and the upper level interconnection network by a bus. The DASH is capable of scaling up to 16 nodes (64 processors) and ideas exist to construct a TeraDASH system consisting of 512 nodes.

A variety of machines can be included in this class and a selection are outlined below. The SGI Origin [Lau97], created by Silicon Graphics Incorporated in 1997, allows the incorporation of up to 512 nodes (made up of 1 or 2 processors) which are connected by a form of hypercube. The IEEE Scalable Coherent Interface (SCI) [Jani90] defines a mechanism for constructing scalable shared-memory multiprocessors using a hierarchy of ring interconnects. The Ultra series of machines from Sun Microsystems [Cha98] allow different interconnects to be used depending on the size of the machine, and also allow small clusters to be created for even larger configurations of machine. The STiNG machine from Sequent [Lov96] uses buses to
connect groups of four processors together and a SCI ring structure to connect the clusters. All of the above systems are NUMA type architectures.

Clustering has also been applied to COMA architectures, for example, the Data Diffusion Machine from the Swedish Institute of Computer Science [Hag92] which has nodes consisting of multiple processors interconnected with a hierarchy of buses, and the KSR-1 [Ram93] mentioned earlier which is also capable of supporting multiple processor nodes.

2.1.3 Hybrid Architectures

There are no set rules to say that a machine must be either message passing or shared-memory, and in fact attempts have been made to take advantage of both styles of communication by constructing machines that contain support for both. The ParaDiGM machine from Stanford [Che91] uses of a hierarchy of buses to construct a shared-memory multiprocessor, but it also contains a switching network that connects the nodes together to allow efficient message passing. The FLASH machine [Kus94], also from Stanford, does not use clustering, but each of the 256 nodes (connected using a 2D mesh) contains a protocol processor that supports both message passing and shared-memory coherency protocols. The SX-4 from NEC [Hem97] contains up to 32 processors in a node sharing a common memory; 16 of these nodes can also be connected using a crossbar interconnect that supports message passing. Yeung et al [Yeu96] describe a clustered system that connects Alewife nodes together with message passing implemented in software.

Other hybrid architectures have been proposed that combine the best features of COMA and NUMA styles. One such hybrid, COMA-F, was proposed by Stenstöm et al [Ste92]; the architecture combined the ability to replicate memory blocks at the main memory level with the notion of a home memory module. This offered an improved performance as the best features of each type of architecture were combined into a single architecture; there was, however, an increase in architecture complexity.
Work has also been carried out on supporting shared-memory on a message passing system and *vice versa*. For example, Vlassov and Thorelli [Vla97] looked at implementing shared-memory on top of PVM message passing, Dwarkadas *et al* [Dwa99] looked at converting shared-memory code at compile time to run efficiently on a message passing architecture and Bernaschi [Ber96] presented a method for implementing message passing efficiently on a shared-memory machine. Packages have also been developed that supply shared-memory on a Network of Workstations, for example TreadMarks [Amz96], which allows shared-memory multiprocessors to be constructed for a fraction of the cost of commercial multiprocessors. Supporting shared-memory through software on a message passing system is frequently referred to as Shared Virtual Memory; an overview of this approach can be found in Iftode and Singh [Ift99].

By providing shared-memory on a message passing architecture, and *vice versa*, a degree of flexibility is provided for programmers, enabling the most appropriate communication model to be used for each application. Implementing shared-memory on top of a message passing system also enables the simpler programming model of shared-memory systems to be combined with the scalability of message passing systems. However, as would be expected, the software implementation's performance level suffers in comparison with its hardware counterpart.

A system constructed that follows this approach is the J-Machine [Noa93] developed at MIT (to study primitive mechanisms for supporting parallel computation). This system contains primitives to allow both message passing and shared-memory communication models. The nodes of the J-Machine consist of a message-driven processor and local memory, and are connected together using a 3D mesh. Message passing primitives are supported by the hardware, with shared-memory being provided through emulation of the processor, cache and directories.
Now that the different types of multiprocessor have been described, the following sections will go into more detail on specific problems associated with constructing shared-memory multiprocessor systems.

2.2 Cache Coherence

Cache coherence is the most important component of a shared-memory multiprocessor and has probably been the most studied and discussed. The cache coherence problem is specific to shared-memory multiprocessors and a variety of solutions and approaches have been proposed in the literature over the past 30 years (some of which will be discussed later in this chapter). Firstly, a brief overview of the cache coherence problem is presented.

2.2.1 The Cache Coherence Problem

The cache coherence problem arises because each processor has its own local cache to provide temporary storage for currently active data, and the address spaces of the processors overlap. It is therefore possible for multiple processors to simultaneously read and write to the same memory location. Cache coherence is the problem of maintaining coherence among all of the caches so that each processor sees the same, up-to-date copy of the data. The set of actions that define the method by which coherence is maintained is commonly referred to as the cache coherence protocol.

Figures 2.5 and 2.6 illustrate this problem. Figure 2.5 shows the two sequences of instructions to be executed by the two processors. Processor 1 reads from location X before updating it, whereas Processor 2 reads from location X twice. Figure 2.6 shows the state of the system at various time intervals when executing the code.
Figure 2.5: Code to illustrate cache coherence

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read X</td>
<td>Read X</td>
</tr>
<tr>
<td>Write X</td>
<td>Read X</td>
</tr>
</tbody>
</table>

Figure 2.6: Illustration of cache coherence

Figure 2.6(a) shows the state of the caches after Processor 1 has issued its read and 2.6(b) shows the state of the caches after both processors have issued a read. Figure
2.6(ci) shows the state of the caches and main memory after Processor 1 has issued a write, assuming that the caches are copy-back (the new value is written to the local cache only). In contrast, Figure 2.6(cii) shows the state of the caches and main memory after Processor 1 has issued a write, assuming the caches are write-through (the new value stored in location X, represented by X', is written to both the local cache and main memory). In both of these cases the result is the same; the cache local to Processor 2 does not contain the new value, so the final read issued by Processor 2 returns the old, incorrect value.

There have been many proposed solutions to this problem, ranging from dedicated hardware that enforces coherence unseen by the software, to compiler-based approaches that guide the processor in deciding if it is safe to cache a value. A selection of solutions is discussed in the subsequent sections.

2.2.2 No Local Caches

The simplest solution is to remove the source of the problem, the local caches. By removing the local caches, the copies of stale data are also removed. All reads will therefore always receive the most up-to-date value, as only one copy of each location exists in the system.

Although this solution is simple, it suffers from serious performance limitations. Firstly, all memory requests have to go through at least one network before being serviced by a memory module, resulting in long memory access latencies. As all memory requests now have to go through the networks and access a limited number of memory modules, a serious bottleneck is introduced, as all requests compete for a limited network bandwidth and for access to main memory.
2.2.3 Broadcast Coherence Protocols

Broadcast protocols are commonly referred to as snoopy protocols, as all of the caches must monitor all network traffic to maintain cache coherence. Snooping protocols have been popular in small-scale bus-based shared-memory multiprocessors, for example the Dragon computer [McC85], Firefly multiprocessor workstation [Tha87], Synapse N+1 [Fra84] and Sequent Symmetry [Lov88].

Snoopy protocols operate by ensuring that all caches and memory modules receive all memory requests. The most common interconnection network used for providing this is the bus. Buses provide efficient broadcasting, as everything connected to them can examine their contents simultaneously. The monitoring of all memory requests enables writes to locations stored in a local cache to be detected and an appropriate action taken.

The snoopy protocols that have been proposed can be broadly categorised by two attributes. The first attribute determines the action to be performed upon the detection of a write. There are two types of action that can be carried out:

- **Invalidate**: This type of protocol invalidates all copies of data held in other caches. For snoopy protocols, an invalidation request is broadcast to all caches, which can then invalidate the copy that they hold.

- **Update**: This type of protocol updates all copies of data held in other caches. For snoopy protocols, an update request is broadcast to all caches along with the new data, which can then update the copy that they hold.

The second attribute describes where data can be fetched from; it also dictates whether updates are sent to main memory as well as to all caches. The decision here is whether to allow other caches to service memory requests, i.e., whether cache-to-cache transfers are allowed.
• **Cache-to-cache transfers**: If caches are allowed to service memory requests, updates need only be broadcast to other caches, with main memory only updated when the data is removed from the caches. When a request is broadcast for data held in a cache, the value is transferred from the cache without the need to access main memory.

• **No cache-to-cache transfers**: By disallowing cache-to-cache transfers, all updates are broadcast to main memory as well as to the other caches. Further requests for data are then serviced by main memory. A request for data that is held in a modified state in a local cache has to be written back to main memory before the data can be returned.

Although snoopy protocols offer good performance for small-scale systems, the need to broadcast requests to other caches limits the scalability. The broadcasts result in too much network traffic for the system to deal with, severely restricting performance for systems with large numbers of processors.

Many snoopy protocols have been proposed, including the Write-Once protocol by Goodman [Goo83] (an invalidate protocol with no cache-to-cache transfers), the Berkeley [Kat85] and Illinois [Pap84] protocols (invalidate protocols that allow cache-to-cache transfers) and the Firefly [Tha87] and Dragon [McC85] protocols (update protocols that allow cache-to-cache transfers). A summary and comparison of the more popular ones can be found in Archibald and Baer [Arc86]. A description of the snoopy protocols included in the shared-memory multiprocessor model described in this thesis can be found in Section 6.3.1.

### 2.2.4 Directory Coherence Protocols

The traditional alternatives to snoopy protocols are directory protocols. Directory protocols remove the need for efficient broadcasting, allowing multiprocessors to be
constructed from networks other than buses, for example, multistage interconnection networks or meshes. This allows machines to be constructed that are potentially more scalable than systems that are constructed from buses using snoopy protocols, for example, the MIT Alewife [Aga99], the SCI [Jam90] and the KSR-1 [Ram93].

Directory protocols maintain information about each memory block and cache line in a single location. Memory requests query the directory before responding, allowing any caches sharing or owning the data to be located and any coherence actions to be taken. The directory is used to direct coherence actions and memory requests to the correct locations, removing the need to broadcast each request to every cache, and allowing other types of interconnection networks to be used to connect processors together.

Directory protocols can also be classified as update or invalidate protocols (as with snoopy protocols) and they can also allow cache-to-cache transfers. Differences between different directory protocols include where the directory is located and how much information is to be stored per memory block. The two main alternatives for directory location are outlined below.

- **Centralised**: The directory is stored close to the memory it is associated with, allowing the directory to be queried when requests arrive at the memory. This organisation is relatively simple to implement, but can be the source of bottlenecks as all requests have to query the same directory.

- **Distributed**: The directory is no longer in a single place; it is distributed amongst the caches. This organisation removes the bottleneck of centralised directories, but is more complex and may have longer query times, as queries have to be forwarded to the appropriate caches.

Agarwal et al [Aga88] introduced the $\text{Dir}_iX$ notation to classify directory protocols. Here $i$ is the number of cache pointers stored per directory entry; these pointers are used to indicate which caches hold a copy of the data. $X$ is either NB (No Broadcast)
or B (Broadcast). In a No Broadcast scheme the number of caches holding a copy of
the data may not exceed the number of indices (invalidation is used to ensure the
number of copies held does not exceed $i$), whereas in a Broadcast scheme the number
of copies may exceed the number of indices, as the protocol changes to broadcast
mode to maintain coherence.

When implementing a directory protocol a balance must be achieved between the
amount of memory required to represent the directory (too much data limits
scalability), directory access time and the number of broadcasts required. This is not
a trivial task and performance may vary greatly between different applications with
different sharing characteristics, requiring different coherence actions.

As with snoopy protocols, many directory protocols have been proposed; a summary
and comparison of the more popular ones can be found in Agarwal et al [Aga88],
Chaiken et al [Cha90] and Lilja [Li93]. A description of the directory protocols
included in the shared-memory multiprocessor model described in this thesis can be
found in Section 6.6.1.

2.2.5 Software Approaches

The techniques outlined above are traditionally termed hardware protocols (as they
are predominantly implemented in hardware). However, coherence protocols may
also be implemented in software. Software protocols can be broadly classified into
two groups:

- **Static**: In static software coherence protocols, coherence activities are planned at
  compilation time.

- **Dynamic**: In dynamic software coherence protocols, the decisions regarding
  coherence activities are made at run time.
Static protocols rely heavily on program analysis at compile time to identify areas of code that may cause the system to become incoherent. How coherence is maintained varies from scheme to scheme; methods include the marking of data (indicating which data should not be cached) and the insertion of special instructions to flush the caches to maintain coherence. Although static protocols remove the need for processors to communicate, potentially allowing more scalable multiprocessor systems to be developed, the program analysis has to be sure that no inconsistencies occur so more coherence actions than are actually required are usually issued, to be on the safe side.

Dynamic software protocols are implemented in the operating system and, as with hardware protocols, maintain coherence at run time. These protocols do not require the preventative measures of static protocols, and allow only necessary coherence actions to be performed as they have information available to them at run time which is not available to compilers.

Software protocols have been popular due to their flexibility, simplicity and low cost. As the protocols are implemented in software, protocols can be written to suit the application that is executing and to switch between different ones dynamically at run time (see Amza et al [Amz99]); they do not require machines with custom hardware, so reducing machine complexity and cost. They have been included in several different machines, including the NYU Ultracomputer [Edl85], the IBM RP3 [Pfi85a], the University of Illinois Cedar Project [Gaj83] and the Munin system from Rice University [Ben90]. Tartalja and Milutinović [Tar96] present an overview of a variety of software protocols.

The sections above outline the basic categories of cache coherence protocols. There are a number of protocols that do not fall into one of these categories, such as hybrid protocols (that attempt to combine the advantages of the different types of protocol). The work undertaken in this thesis considers some of the more popular snoopy and directory protocols that have been proposed.
2.3 Synchronisation

The techniques described in the previous section maintain the multiprocessor in a coherent state; however the order in which the processors execute their instructions is not determined. This means that different processors could attempt to update the same memory location at the same time, resulting in an uncertain outcome for the final value of the location. Synchronisation enables access to shared locations by different processors to be controlled.

To implement synchronisation in a multiprocessor system a level of hardware support is required. Hardware primitives are needed to provide operations that atomically read, modify and write to a memory location. The more common primitives provided usually include test-and-set, fetch-and-add and compare-and-swap. These allow memory locations to be inspected and updated without interference from another processor.

These hardware primitives are used by software to provide a collection of synchronisation primitives, for example, spin-locks and barriers.

2.3.1 Spin-locks

Spin-locks are commonly used to provide mutual exclusion, i.e., to ensure that a limited number of processors, usually one, accesses a particular section of code at any one time. Spin-locks can be implemented in numerous ways, but in general they require a shared memory location and an atomic memory operation, for example, test-and-set. When a processor wishes to enter a critical section of code protected by spin-locks, the shared variable is inspected. If the shared (lock) variable is 0 (indicating no other processor is in the critical section) the processor sets it to 1 before continuing into the critical section. If the value of the shared variable is 1 (another processor is already in the critical section) the processor must wait, continually inspect the shared location waiting for it to be set to 0 by a different
processor before proceeding. When the critical section is exited the lock is reset to 0, allowing a different processor to proceed. The atomic operations ensure that only one processor can see the value of 0 and set it 1. This form of implementing synchronisation is commonly referred to as *busy-waiting*, as the processor must continually check the spin-lock variable.

### 2.3.2 Barriers

Barriers are used to ensure that a group of processors have all reached a particular point in the program before any of them are allowed to proceed. Barriers can be implemented in a variety of ways but the general operation requires shared memory locations (a counter and a flag) and an atomic memory operation, for example, fetch-and-add. When a processor arrives at a barrier it decrements the shared variable using fetch-and-add. If the result of the decrement is 0 (all processors have arrived) the shared flag is set to 1, indicating to all other processors that the final processor has arrived; otherwise the processor waits for the shared flag to be set to 1 by another processor. This is a busy-waiting implementation of a barrier.

Synchronisation operations can cause serious bottlenecks in the multiprocessor system. The bottlenecks arise from multiple processors reading and modifying the same memory location, and this problem becomes progressively worse as more processors are added to the system. Efficient synchronisation is therefore critical for reasonable performance and this has resulted in many different attempts to provide efficient implementations. Mellor-Crummey and Scott [Mel91] presented a survey of the more common approaches proposed for synchronisation of shared-memory multiprocessors. A discussion of the synchronisation primitives implemented in the multiprocessor model described in this thesis can be found in Section 6.2.3.
2.4 Models of Memory Consistency

Cache coherence ensures that processors see a consistent view of memory, but this is not enough to guarantee correct execution of a program. Cache coherence does not determine when a processor sees the new, updated value of a memory location, only that the system will be become coherent eventually. As processors communicate through shared variables, reads will frequently follow writes to a shared location, and synchronisation alone is not enough to guarantee that the right value will be returned by a read to a recently written location. The delays in the system may cause two consecutive requests to the same locations by the same or different processors, giving an uncertain result (as the second request may overtake the earlier one and complete first).

Consequently, an ordering has to be imposed on the system to ensure that, in critical situations, earlier instructions complete before later ones. Memory consistency models are used to specify the order in which instructions from the same and different processors may be completed.

As with cache coherence and synchronisation, memory consistency models have been studied in detail for the last 20 years. In this time many approaches have been proposed, including sequential consistency [Lam79], weak ordering [Adv90] and release consistency [Gha90].

2.4.1 Sequential Consistency

The strictest model for memory consistency is sequential consistency, defined by Lamport [Lam79] as:

"[A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential..."
order, and the operations of each individual processor appear in this sequence in the order specified by its program."

This means that all instructions executed by all processors must be completed in a particular sequential order. This control of all instructions appears over strict, as the instruction ordering is only important for those which access shared locations that are read from and written to. The order in which instructions that access non-shared or read only shared locations complete has no impact on the correct operation of the program.

This model maintains consistency in a manner that is invisible to the programmer and programs execute without any unpredictable behaviour. However, sequential consistency has a performance disadvantage, as instructions must be completed in order, meaning that common architectural features such as write buffers cannot be used. Write buffers allow a write to be placed in a buffer and performed by the cache without the need for the processor to stall. This allows the processor to continue execution before the write has completed, including the issue of memory requests that may be satisfied by the cache. This is out of order completion of memory requests is not allowed when the sequential consistency model is used.

To overcome this performance disadvantage, other models of memory consistency have been proposed. These attempt to trade-off the simplicity of the programming model with performance; two of these models, weak ordering and release consistency, are discussed below.

2.4.2 Weak Ordering

Weak ordering relaxes the strict order imposed by sequential consistency, allowing most instructions to be completed out of order. In this model the responsibility for ensuring correct and consistent operation is passed to the programmer. Any instruction is allowed to complete before any other issued instruction, unless specifically stated by the programmer that this would be incorrect.
The control is performed through the use of synchronisation operations. In the weak ordering model of memory consistency, synchronisation operations are not executed until all earlier instructions have completed, and no further instructions are allowed to start until the synchronisation primitive has completed. This restriction at synchronisation points is equivalent to sequential consistency. Therefore weak ordering imposes no ordering among normal instructions, but does impose sequential consistency at synchronisation points.

To take full advantage of the performance that is offered by weak ordering, several architectural features are required to ensure that the processor is allowed to continue execution without waiting for the result of a memory request; these include write buffers and caches that support non-blocking reads.

Although weak ordering offers a potential increase in performance, it is now the responsibility of the programmer to ensure consistent operation through the use of synchronisation operations. If not used carefully, these could become bottlenecks and reduce system performance.

2.4.3 Release Consistency

Release consistency models further extend the ideas used in weak ordering. Release consistency also allows for instructions to be completed in any order, except at synchronisation points. However, in this model a distinction is made between the types of synchronisation operation being performed, and the model defines which operations need to be controlled. This is unlike weak ordering, which says that all synchronisation operations need to be controlled.

Release consistency splits the synchronisation operations into two categories, acquire and release. Particular combinations of reads or writes preceding or following acquires or releases are allowed to proceed and complete in any order. However, certain combinations are not allowed to complete out of order and must
follow the sequential consistency model, namely writes (after acquires or releases), reads (before and after acquires), and also any combination of synchronisation operations; these all must conform to the sequential consistency model.

Release consistency can offer potential performance advantages over sequential consistency and weak ordering, but again the responsibility of ensuring correct and consistent operation lies with the programmer. Careful use has to be made of synchronisation operations to ensure the performance potential is realised.

Many other models of memory consistency have been proposed, discussed and compared at length in the literature, including total store order, processor consistency and partial store order, for example, Adve and Hill [Adv96] and Dwarkadas et al [Dwa93]. A discussion of the consistency models included in the multiprocessor model described in this thesis can be found in Section 6.2.4.

This chapter has aimed to provide a background to the field of multiprocessor systems, in particular shared-memory multiprocessors, and to present some of the problems that must be addressed in order to design a machine that operates correctly. The next chapter will discuss a selection of the approaches and environments that have been proposed to evaluate the different solutions to these problems.
Chapter 3

Multiprocessor Modelling and Simulation

This chapter describes the various techniques that have been used to explore the possible design trade-offs and to evaluate the performance of multiprocessor systems. Firstly, performing experiments using actual hardware and systems will be discussed, followed by an outline of the mathematical and analytical techniques that have been proposed to estimate the performance of multiprocessors. Finally, multiprocessor simulators and simulation environments that have been developed will be discussed.

3.1 Evaluation using Multiprocessor Systems

As described in the previous chapter, there are several types of multiprocessor architecture, with many possible different implementations. There are also many possible solutions, implemented in either hardware or software, for the problems highlighted. One approach to evaluating the different design trade-offs is to implement and execute a possible solution on a real system.
By implementing a complete system, results can be obtained by running benchmarks and complete applications. These results are very accurate as they include all system overheads, such as operating system and disk access delays. The complete results can also be obtained relatively quickly, as the actual hardware is used as opposed to a simulation of the hardware.

The downside to using actual systems is that they are inflexible, reducing the number of design alternatives that can be evaluated easily. It is difficult and time consuming to change the major architectural characteristics of a system, for example, changing the interconnection networks, changing the grouping of processors or converting a NUMA to a COMA style architecture. It is also difficult to change the cache coherence protocol, the synchronisation primitives and the memory consistency model, as they may require different hardware support.

The cost of such systems is also a limiting factor for this approach, as not all can afford to purchase a multiprocessor system with which to experiment. To overcome the cost restriction, networks or clusters of workstations are a popular choice to explore different implementations. Bal et al [Bal98] used this approach to examine and optimise the performance of applications on workstations connected over a wide-area network. Parsons et al [Par97] used TreadMarks and an eight workstation system with either a 100 Mbs fast ethernet or 155 Mbs ATM interconnection network to investigate parallel application performance.

The restrictions of using multiprocessor machines to explore and evaluate design trade-offs have led to a variety of techniques and approaches being proposed that do not require expensive hardware. A selection of these will be discussed in the following sections.
3.2 Mathematical and Analytical Models

Analytical models have been used for many years to evaluate various aspects of multiprocessor systems, from individual components (for example caches and interconnection networks) to complete shared-memory multiprocessors (including cache coherence). Mathematical models and techniques are used in an attempt to approximate the behaviour of a system and predict its performance. Some of the models that have been proposed are outlined below.

The two components of a multiprocessor that have been investigated the most are caches and interconnection networks, as a good estimation of their performance can be achieved by considering them in isolation. By doing this, the number of options to be considered in the final system are reduced, as the designs of the cache and interconnection network have been chosen.

Agarwal et al [Aga89] presented a detailed analytical cache model that covered the effects of startup, non-stationary behaviour (changing working set), interference and multiprogramming. The cache was specified by the number of sets, associativity and block size and was driven by parameters that were extracted from program traces.

Various interconnection network models have been proposed to study a range of different network topologies. Patel [Pat81] gave a performance analysis of delta networks ($(a^n \times b^n) - n$ stages of $(a \times b)$ crossbar switches) and crossbars used in message passing machines (no shared data) that were driven by workload models. Ould-Khaoua [Oul98] used queuing models to study a hypercube, as well as 2D and 3D torus under uniform traffic. Agarwal [Aga91] presented an analytical model to study different k-ary n-cube interconnection networks for shared-memory multiprocessors along with the effects of switching and wire delays. Fu and Chau [Fu98] gave an analytical model for cyclic-cubes. Willick and Eager [Wil90] used queuing networks and mean value analysis to model synchronous, packet switched networks with buffers at each stage, using a workload model as input. Many of these
models have parameters to allow exploration of the design space; the more common parameters include the number of processors, the number of memory modules and network configuration (for example, number of switches per level and number of levels).

Although the above models give an estimate of the performance of particular components, they do not give an indication of the performance of the whole system. The complete multiprocessor system may also generate a different workload from the one used to decide upon the components' design. To overcome this, analytical models of complete systems have been proposed.

Analytical models have been proposed for both message passing and shared-memory multiprocessors. Patel [Pat82] studied message passing machines, consisting of processors and caches connected to memory modules using an \((n \times m)\) multistage network. Giloi et al [Gil98] used deterministic and generalised Petri nets to model distributed message passing machines with a processor, cache and memory module per node. Yang and Bhuyan [Yan88] also studied a message passing system that connected the processors to the memory modules with multiple buses.

Shared-memory systems were considered by Vernon and Holliday [Ver86] who modelled the performance of snoopy cache coherence protocols on systems consisting of processors and caches connected to main memory via a bus using generalised Petri nets. Dubois and Wang [Dub88] used Markov chains to study cache coherence protocol performance; Stenström [Ste89] also studied cache coherence protocol performance for multistage networks. Adve et al [Adv91] compared the performance of hardware and software cache coherence protocols using mean value analysis and queuing networks, but did not consider the interconnection network and limited the cache line size to one word. Sorin et al [Sor98] modelled an entire shared-memory multiprocessor, except for synchronisation.
The most well known analytical model for parallel systems is the Bulk Synchronous Processing (BSP) model [Val90], which views a machine as a set of processor/memory pairs with a global communication network and includes a method for synchronising the processors. A BSP program consists of a sequence of super-steps; each super-step has three phases: (1) each processor performs a number of computations on local data, (2) the processors communicate data to other memories and (3) all of the processors perform a barrier synchronisation. This model can be used to develop parallel applications and assess their performance. Various delay parameters can be specified to model network and memory access delays. The LogP [Cul96] and LogGP [Ale95] models have been proposed to extend the BSP model, by taking into account communication costs (latency, overhead and gap between consecutive messages) and long messages.

Analytical models have also been proposed to take into account the effects of clustering processors together to form a system with multiprocessor nodes. These include Hua et al [Hua91] and Lundberg and Lennerstad [Lun98]. Hua et al presented a model of a system with sharing within a node, but not between nodes, with parameters to change the configuration of the clusters and network bandwidth. Lundberg and Lennerstad presented a model of a message passing system that allowed different cluster configurations to be evaluated.

The discussion of the various types of analytical model shows that there are many different kinds of model, each with their own particular area of focus, although some models have been proposed that try to model the entire system in detail. Analytical models provide a much more flexible approach to multiprocessor design exploration than experimentation with real hardware. The models can be parameterised to allow easy exploration of the alternatives; different models have included different input parameters, including the number of processors and memory modules and the interconnection network topology and system delays, for example, network latency and memory access time. The models are usually driven using workload models instead of benchmarks or real applications, producing results that are not as accurate or reliable. Accuracy is also a problem within the model itself, as mathematical
models are used to approximate the behaviour of actual hardware which, although they produce results relatively quickly, do not always ensure accuracy. The most accurate of the models outlined above was the one produced by Sorin et al [Sor98] which produced results that were within 12% of those achieved by simulation.

Another approach used to evaluate multiprocessors and predict their performance is simulation, which attempts to improve on the accuracy of analytical models; this will be discussed in the next section.

3.3 Simulation of Multiprocessor Systems

Simulation has become a popular approach for predicting the performance of computer systems. The basic idea is that code is written that represents the behaviour of the system to be evaluated. When executed, the simulation behaves in a similar manner to the system being represented, enabling experiments to be performed and the performance of the modelled system to be estimated. Several approaches to creating simulations have been proposed; a selection of these will be discussed in the remainder of this section. Firstly, various simulation environments that have been used will be outlined. This will be followed by a discussion of the different types of multiprocessor simulation that have been created and used to study various aspects of multiprocessor systems.

3.3.1 General Simulation Environments

There have been many different simulation environments developed; some of the better known of these will be discussed, along with their applicability to multiprocessor system simulation. The systems discussed are used in a variety of application domains; systems aimed at the simulation of multiprocessors are discussed in Section 3.3.2.
SimOS [Ros95, Ros97] was developed at the Computer Systems Laboratory at Stanford University and was originally intended to aid in the study of the execution behaviour of operating systems. SimOS simulates the complete hardware and software of the target system, in enough detail to enable operating system code to be executed along with the application code. SimOS has three modes of operation. The first, emulation mode, simulates only the components of the hardware required to execute translated instructions, i.e., no caches and I/O modules. This mode does not include a timing model but runs at high speeds. The second mode, rough characterisation, is slower than emulation mode but provides rough timing models for instruction execution, memory stalls and I/O behaviour. Unified caches are modelled at this level, but misses are assigned a uniform delay instead of modelling the complete memory hierarchy, a basic I/O timing model is included. The final mode, accurate mode, models the entire system and is much slower than the other two. SimOS is capable of switching between the three modes, enabling a simulation to be executed in emulation mode until areas of interest are reached, when the simulation is switched to rough characterisation or accurate mode. SimOS includes a model of a MIPS processor that can be included in simulations. The processor model achieves high levels of performance by using the technique of binary translation (developed in the Embra simulator [Wit96]) which converts blocks of instructions from the application into code sequences that implement the effects of the original instructions on the simulated processor.

SimOS supports memory system models for a bus-based multiprocessor with uniform memory access times, a simple cache coherent NUMA memory system, and a cycle accurate simulation of the Stanford FLASH memory system. The obvious advantage of SimOS is the speed of two of the modes; however these modes cannot be used when studying the memory system of a multiprocessor, as memory accesses are assigned a uniform delay in the quicker simulation modes. These modes can be used to get the system into a steady state, i.e., boot up the operating system and initialise the application, before switching to more accurate modes to model the memory hierarchy. Simulations are also limited to using MIPS processors as they are included in the environment; although this is not a serious limitation, it does
restrict design alternatives, as different processor characteristics cannot easily be examined. As the entire system is simulated, including the operating system, extremely accurate results can be obtained (assuming the model is also very accurate).

The Ptolemy system [Dav99] is collection of tools to aid the design, modelling and simulation of concurrent systems, with a focus on embedded systems that contain mixed technologies, for example, analog, digital and mechanical components.

Recently Ptolemy II [Dav99] has been released; it is a based on the original Ptolemy system but implemented in Java. The experiences gained from the Ptolemy system have enabled the new system to be implemented in a more modular structure, allowing the system to be extended more easily.

Ptolemy’s most important feature is its ability to simulate a complete system from components simulated with different simulation systems. The different simulation systems are referred to in Ptolemy as models of computation; these include communicating sequential processes, continuous time, discrete event, finite-state machines, process networks and synchronous dataflow. Ptolemy has been used for many simulations but with an emphasis on embedded system design, including hardware/software codesign.

POLIS [Chi96] is a simulation environment that is targeted towards real-time control/embedded systems that are based on microcontrollers and semi-custom hardware components. The environment is based on Co-Design Finite State Machines and implements the complete design flow from high level specification and partitioning to simulation and synthesis. This system is not aimed at simulation of multiprocessor systems and consequently provides no support for it.

Multiprocessor simulation with these systems, although possible, requires significant work from the designer as no specific support is provided to aid in the development
of these simulations. Systems aimed at the simulation of multiprocessors, or specific aspects of a multiprocessor system are discussed in the next section.

As well as simulation environments, a large number of simulation languages have also been developed. The most common type of simulation language used for computer system simulation is discrete event. This is a simple approach to simulation that maintains a list of outstanding events that are processed in a chronological order. Events are used in many ways, for example, to communicate between entities and to reactivate idle entities. Examples of simulation languages include SIM++ [Jad91], CSIM [Sch94], and Synchronous C++ [Pet98]. Simulation languages provide no support for any particular model; they provide the mechanisms through which simulations may be constructed. They usually supply functions that allow components to communicate, for example send and receive, as well as a notion of time, allowing constructs such as hold to be used. These languages form the basis for the different environments discussed, as well as specific multiprocessor simulations described later; they do not however provide the graphical interface and component library structures of some of the environments.

3.3.2 Multiprocessor Simulators

This section describes systems that have been developed specifically for simulating multiprocessor systems. The method by which the simulation generates/uses memory references is one of the main differences between the different simulators. This will be referred to as the method by which the simulation is driven. There are four main methods used, distribution-driven, trace-driven, application-driven and execution-driven. Each of these will be discussed in the following sections. Along with each type, examples of relevant multiprocessor simulators will be given.

The multiprocessor simulators developed can be split into two categories, multiprocessor simulation frameworks (those that allow different machines to be specified and evaluated) and dedicated simulators (those that have been developed to investigate a particular architecture and offer only limited flexibility).
(i) Distribution-Driven Simulation

In a distribution-driven simulator each of the processors in the system is replaced by a component that generates random memory references at random intervals. The distribution of the memory addresses and intervals is tailored to match the required workload. The advantages of this approach are that it is simple and easy to implement (assuming a mathematical model exists that corresponds to the required workload) and does not increase the overall simulation time significantly. The models created to drive the simulation are usually parameterised to allow the workload to be modified easily. Mathematical models can also be used to easily express non-realistic workloads that can examine the operation of the machine in unusual circumstances, for example, every memory reference is to a remote memory module with a constant delay between consecutive references.

However, the disadvantage of using a distributed-driven approach is that it is extremely difficult to express the complex pattern of memory references of a multiprocessor machine with a mathematical model. Not only is the specific pattern of addresses difficult to model, but the timing between these references is also extremely difficult to express mathematically. This limits the accuracy of the simulation and effectiveness of the results obtained when evaluating multiprocessor architectures. One approach that has been taken to increase the accuracy of this method is to analyse parallel applications and use the characteristics obtained to specify a more accurate model. An example of this is illustrated by work performed by Brorsson and Stenström [Bro94] who looked at expressing the degree and type of sharing of parallel applications and using these characteristics to generate references for a shared-memory multiprocessor simulator.

Distribution-driven simulation has been an approach used to study interconnection networks. Pfister and Norton [Pfi85b] used a simulation driven by a uniform distribution (with hot spots overlayed) to study omega networks with four to sixty four processors. Duato and Malumbres [Dua96] used a network simulator driven by
generated workloads to compare the performance of hypercubes and 2D meshes with 256 nodes.

There have also been several complete multiprocessor simulators that have been developed using this approach. Lovett and Clapp [Lov96] developed a simulation of the STiNG multiprocessor from Sequent Computer Systems, Inc. which was driven by references generated from a distribution obtained from profiles of the TPC database benchmarks programs (probabilities of cache misses, invalidations and I/O events are specified). Omran and Aboelaze [Omr94] used CSIM to create a simulation to test a cache coherence protocol for multistage networks, which was driven by references generated by a workload model. This simulator offered a small number of architectural parameters to be changed, including the number of processors, memory access time and cache access time. Yang et al [Yan92] used a simulation driven by an artificial workload model to study a new snoopy cache coherence protocol for hierarchical bus architecture. Nanda and Bhuyan [Nan93] also used this approach to study cache coherence in multistage and multiple bus networks; the simulation provided several architectural parameters that allowed easy experimentation, including number of processors, memory access time and number of levels in network hierarchy. Archibald and Baer [Arc86] compared the performance of several well known snoopy cache coherence protocols. Finally, Grujić et al [Gru96] modelled four different architectures (based on the DASH, SCI, DDM and KSR-1), driven by a synthetic workload to compare their performance. The architectures could vary the number of clusters, the number of processors per cluster, (maximum of four, however total number of processors must be constant), processor speed, cache access time and cluster bus cycle time.

Although this approach to driving a simulation has been relatively popular for developing a simulation to study a particular architectural feature, it has not been well utilised in multiprocessor simulation frameworks.
(ii) Trace-Driven Simulation

In a trace-driven simulator a trace is generated by executing a workload on a similar machine and tracing the references made; this trace is then used to drive the simulation. Holliday and Ellis [Ho192] viewed trace-driven simulation as having the main phases of trace collection, trace reduction and trace processing; each of these will be discussed below.

Trace collection is the process of determining the sequence of memory references for the required workload. There are several methods for collecting traces, including hardware probes, microcode modifications and instruction set emulators. Hardware probes can be used to monitor the traffic on the system buses and record the references executed by the workload. Alexander et al [Ale96] used hardware monitors to collect traces in their study of the impact on performance of different cache parameters. The microcode of the machine executing the workload can be modified to output all the memory references to the trace file. Sites and Agarwal [Sit88] presented the ATUM-2 system, capable of generating traces using this approach for workloads executed on a VAX 8350. Instruction set emulators (a processor simulated in software) can also be modified to output memory references (in a similar manner to the modified microcode). For a discussion of the different trace collection techniques see Uhlig and Mudge [Uhl97].

To obtain a trace file with enough references in it to obtain accurate results requires a large amount of storage. Trace reduction techniques can therefore be used to remove redundant and unnecessary references. These techniques include compression, filtering and sampling, a discussion of which can also be found in Uhlig and Mudge [Uhl97].

Once the two phases of trace collection and trace reduction have been completed, the result is a manageable trace that can be used by the third and final phase. This phase, trace processing, is the actual system simulation that uses the created trace to mimic the system behaviour. In trace-driven simulation, each of the processors in the
system is replaced by a component that is capable of reading the references from the trace file and issuing them to the rest of the system. These components must also be capable of dealing with the data returned as a result of issuing memory references.

There are several problems with using traces to drive simulations. Firstly, collecting a complete and detailed address trace for the application to be studied, suitable for use on the system that is to be simulated, is difficult. This is because the workload must be executed on a machine that is not the same as the system that is to be simulated. The sequence of memory references generated by executing the workload may therefore be different from the sequence that would be generated by the simulated system. There are also problems associated with generating traces for multiprocessor systems, for example, either a multiprocessor with the appropriate number of processors, or a method of generating a trace for one processor and using this to generate a trace for all the other processors in the system is required. Secondly, for the trace to be useful, a large number of references must be included, resulting in a large trace, which could require many gigabytes of storage space. Processing this large file can be very time consuming, as each processor must retrieve each memory reference from a file. Trace files also reduce the effectiveness of a simulation used to study the effects of varying the number of processors in the system, as a different set of trace files is required for each configuration simulated. Finally, if only memory references are recorded in the trace, no useful application results can be obtained, for example execution time, as the time between the references is not stored.

Despite these disadvantages, traces have been used to drive many simulations. The main reason for this is that the component used to process the traces in place of the actual processors is much simpler, making simulations easier to develop. A complete simulation of a processor capable of executing the application is very complex and would increase significantly simulation execution times.

Trace-driven simulation has been a popular technique used in the simulation of memory systems. Smith [Smi82] used trace-driven simulation to study the effects of
a wide range of cache parameters, with multiprogramming modelled by switching input traces at periodic intervals. The trace files contained one million references. Alexander et al [Ale96] studied cache performance with traces that included operating system references, interrupts, task switches, prefetching effects and I/O activities as well as the application being evaluated. Ewy and Evans [Ewy93] looked at the effects of second level caches using the Shadow system. The traces are generated dynamically (removing the need for large amounts of storage) and fed directly into the simulator allowing much larger traces to be used (experiments performed used up to 210 million references). Finally, Dinero [Ed196] is a well-known, freely available, trace-driven cache simulator that simulates a memory hierarchy with multiple levels of caches with a variety of different parameters. However, Dinero does not include timing information (it is mainly concerned with hit/miss rates) nor does actual data move around the hierarchy, only references.

Trace-driven simulation has also been used to model complete multiprocessor systems. Eggers and Katz [Egg88] modelled a shared-memory multiprocessor with between 5 and 12 nodes (containing a processor and a cache) with cache coherency maintained by either the Berkeley or Firefly protocols (see Section 6.3.1 for a discussion of these two protocols). The traces used for each processor in the simulation contained 300,000 references. The simulation developed has also been used to compare the performance of new protocols (see Eggers and Katz [Egg89]) that have been proposed to overcome the weaknesses of the Berkeley and Firefly protocols. Tullsen and Eggers [Tul93] developed a simulation of a bus-based shared-memory multiprocessor (with between 9 and 12 processors, using the Illinois protocol to maintain coherence) to enable five prefetch algorithms to be evaluated. Traces were generated using MPTrace [Egg90] on the Sequent Symmetry system and contained 2 million references per processor. Chaiken et al [Cha90] used trace-driven simulation to compare different directory cache coherence protocols.

Despite the simplified simulation approach that using traces offers, its many shortcomings have resulted in a need for a different approach for simulating
multiprocessor systems. A popular approach, application-driven simulation (which is used to remove the need for large trace files) is discussed next.

(iii) Application-Driven Simulation

Application-driven simulation is at the other end of the complexity spectrum to distribution and trace-driven simulation. The simple processor model of distribution and trace-driven simulation is replaced by a simulation of a processor that models the operation of the complete instruction set. The simulated processors are then used to execute the workload, with the memory references being passed on to a memory simulator.

The advantage of this approach to multiprocessor simulation is the accuracy of the results produced. The execution of the workload on the simulated processor produces results that are not only based on the sequence of memory references, but also on the correct timing of these references. The improved accuracy is achieved at the expense of simulation execution time. The detailed simulation of each of the processors in the multiprocessor executing the workload can take a long time to execute when compared to the other forms of simulation.

A further limitation of application-driven simulations is that only the processor modelled can be used in the experiments; changes in the processor can be difficult and time consuming to implement. The accuracy of application-driven simulation has proved appealing to developers, however, with many simulations being produced that use this technique. A selection of multiprocessor simulation frameworks will be outlined next, followed by a discussion of other simulations that have been implemented.

RSIM [Pai96] is an application-driven simulator for multiprocessors that exploit instruction level parallelism, which models the processor, memory system and interconnection network in detail. The modelled processors are capable of executing
code generated directly by a compiler. The interconnection network simulator has been developed from the Rice Parallel Processing Testbed [Cov88].

RSIM has been used in many studies of multiprocessors. Pai et al [Pai96] compared release and sequential consistency models for write-through and copy-back caches. The systems evaluated contained 8 or 16 processors, with hardware prefetching and speculative loads. Gniady et al [Gni99] used RSIM to model an eight-node (one processor per node) distributed shared-memory system. This was used to compare sequential consistency (executed on processors that exploited instruction-level parallelism) to release consistency (executed on simple processors). Pai et al [Pai97] used RSIM to assess the impact of instruction-level parallelism on shared-memory multiprocessors with 8, 16 or 32 processors.

The CacheMire Testbench [Bro93] was developed at Lund University, Sweden, as an environment for conducting performance evaluations of shared-memory multiprocessors. CacheMire consists of a simulator and a programming environment. The simulator contains three main components, a highly optimised instruction set emulator for the SPARC instruction set, a memory model and a multiprocessor framework. The instruction set emulator is responsible for executing the application code, the memory model simulates the behaviour of the memory system and the multiprocessor framework coordinates the operation of multiple instruction set emulators (one for each processor in the system to be evaluated). The programming framework provides run-time libraries and routines, for example I/O and memory allocation. Each component is separate and can be replaced easily, allowing different memory hierarchies to be evaluated with a minimum of effort.

CacheMire has been used to evaluate various aspects of shared-memory multiprocessors. Brosson et al [Bro93] discussed a simulation of a cache coherent NUMA architecture, similar to the DASH, that was developed to aid the study of write-invalidate and write-update cache coherence protocols. Barroso and Dubois [Bar93] used CacheMire to generate address traces for multiprocessors with 8, 16 or 32 processors using a snoopy or directory protocol, which were fed into a simulation.
of different ring interconnects written using CSIM. Stenström et al [Ste97] used CacheMire to evaluate possible performance advantages of release consistency, sequential prefetching, migratory sharing detection and an update/invalidate hybrid protocol on a multiprocessor with 16, single processor nodes.

MINT [Vee94] is a system developed from a collaborative effort between the University of Rochester (New York) and the University of Copenhagen to aid in the construction of multiprocessor memory hierarchy simulators. MINT uses instruction interpretation to execute the workload, although native execution is sometimes used to speed up execution times (when the instruction to be interpreted is similar to an instruction on the host machine). The system links together the compiled workload, simulation libraries (including a simulation model of a MIPS processor capable of running UNIX executables) and a multiprocessor memory system simulator.

The MINT system has been extended to form the SMART system [Gab97], which allows system level events to be studied, including process switching and task migration. The memory references generated by the processors in MINT are passed onto SMART. The multiprocessor architecture model used to study these events allows several parameters to be changed, including the number of processors, cache size, associativity, block size, replacement policy, bus width and coherence protocol.

An instruction level simulator was used by Anderson and Baer [And93, And95] to explore the benefits of using a customised cache coherence protocol in a hierarchical bus based clustered multiprocessor. Zucker and Baer [Zuc92] also used an instruction level simulator to model a multiprocessor machine connected by an omega network to study the effects of different memory consistency models. Landin and Karlgren [Lan97] used the Silics [Mag95] instruction set and cache simulator to study the effects of clustering in COMA architectures with 16 processors with clusters of 1,2 or 4 processors.

Application-driven simulation produces accurate results, but the execution times needed to produce them can be very long. To overcome these long simulation
execution times a new approach is required. The next section discusses execution-driven simulation, which has been proposed as a faster, although slightly less accurate method for simulating computer systems.

(iv) Execution-Driven Simulation

In execution-driven simulation the workload is executed on the host machine. When execution reaches a memory reference it is supplied to the system simulation and the workload may be suspended while the simulator deals with the reference.

The main advantage of this approach is that most of the instructions of the workload are executed using the processor of the host machine, enabling them to be completed quickly. The only instructions not executed in this manner are those that involve the memory system in some way, for example, loads and stores (on a shared-memory machine) or sends and receives (on a message passing machine). Instead of executing these directly on the host machine, calls to simulation are executed instead. By executing workloads as part of the simulation, it is no longer necessary to generate and store large traces. The workload can also be written in such a way that it can adapt to the number of processors in the system at run time, unlike traces which require a different set of files for each configuration of the multiprocessor simulation. By generating the references at run-time, the need to use large trace files is removed, allowing simulations to be run with very large numbers of references.

During the remainder of this discussion of execution-driven simulation of multiprocessors, shared-memory multiprocessors (using loads and stores) will be used, but the arguments made can be equally applied to message passing machines (using sends and receives).

The problems with execution-driven simulation revolve around the calls to the simulator, for example, how calls are inserted into the workload code and how calls are synchronised with the simulation.
The benchmark or application used as a workload is usually written in a high level language, for example, C or Fortran. This code can be compiled on the host machine and then executed. Many solutions have been proposed for inserting calls to the simulator where appropriate. These include the method used by PROTEUS [Bre91] that required the source code to be annotated before compilation, and that used by the TangoLite system [Her], which automatically annotates the compiled code with calls to read and write functions that must be supplied by the simulation.

The second problem, synchronising the workload execution with the system simulation, affects when the calls to the simulator are made. Most of the instructions are executed on the host machine, which does not affect the simulation time (the predicted execution time of the simulated system). The result of this is that loads and stores issued by the simulated processor will be issued immediately after the previous one has finished. To make the system execute correctly, the simulated processors must be stalled before each memory reference. The length of time for which the simulation is stalled represents the time it would take the processor of the simulated machine to execute the instructions between the two references, i.e., those that were executed on the host machine. This problem can be solved for simple processors by recording the instructions executed between the memory references and then estimating the time that each instruction would take on the processor used in the target machine. For more complicated processors, for example, processors that use instruction level parallelism (speculative loads, non-blocking loads and out-of-order execution) this approach is not sufficient. The time taken can be estimated but not accurately recorded; to deal with these processors, more complex models are needed which attempt to predict the execution time of a set of instructions. Durbhakula et al [Dur99], for example, used a timing simulator to predict the time between references.

Once these two problems have been overcome, execution-driven simulation offers good performance, as most of the instructions are executed on the host machine, with increased flexibility and accuracy. The flexibility is achieved because the code is executed each time the simulation is run, allowing it to adjust dynamically to the
number of processors in the system. The accuracy is gained by running the complete application/benchmark and taking into account all of the instructions (unlike most trace-driven simulations that are only interested in memory references).

Execution-driven simulation has been used to investigate parts of a multiprocessor system. Jouppi [Jou93] used execution-driven simulation to study the effects of cache write policies with large numbers of references. The Reconfigurable Architecture Workbench [Lig97] developed at the Georgia Institute of Technology, was used to evaluate different interconnection networks for a 4096 node SIMD machine. This system allows several architecture parameters to be altered, for example, topology, bandwidth and latency. Bhuyan et al [Bhu98] used PROTEUS (discussed later in this section) to study the effect of switch design on the performance of cache coherent shared-memory multiprocessors that used a multistage interconnection network.

Many execution-driven simulations and simulation frameworks have been developed to study multiprocessor systems. A selection of simulation frameworks will be discussed first, followed by an outline of other multiprocessor simulations.

PROTEUS [Bre91] is an execution-driven multiprocessor simulation framework specifically aimed at the simulation of MIMD multiprocessor systems. It models nodes containing a processor, cache chip (for cache coherence), a network chip and memory (split into private and shared). PROTEUS provides the simulation kernel and modules for implementing shared-memory, as well as the processor, cache and network modules. These modules require the designer to provide a small number of function definitions, for example, the network module requires the send, route and receive functions to be provided. This allows different networks to be included in the simulation by changing the definitions of these functions. The function definitions can also be used to implement different levels of complexity, for example, the cache module could contain a complete implementation of a coherence protocol, no coherence protocol code, or it could use the physical memory of the host machine to retrieve the data and wait for a uniform period of time before returning.
the data to the processor. The environment also provides mechanisms for allocating shared-memory and message passing which are used to annotate the application source code before compilation. The compiled application is then linked to the multiprocessor system simulation before execution.

PROTEUS has been used to construct several multiprocessor simulations, including one of the nCUBE message passing machine and the Alewife shared-memory machine. PROTEUS offers several advantages; it is flexible, allowing architectural freedom (designers are able to change the implementations of the cache, network, memory and processor components), offers reasonable performance and accuracy and support is provided that enables applications to be annotated, compiled and executed using the simulation.

SPASM (Simulator for Parallel Architectural Scalability Measurements) [Siv99] is an execution-driven simulator written in CSIM capable of simulating parallel applications on shared-memory and message passing multiprocessors. When simulating a shared-memory multiprocessor, a preprocessor is used to insert code into the application to switch to the simulator on a shared-memory reference. Sends and receives inserted by the programmer for a message passing multiprocessor also switch to the simulator. In both cases, the compiled assembly code is augmented with cycle counting instructions, which are used to count the number of cycles between switches to the simulator.

The SPASM simulator allows a number of parameters to be varied, including the number of processors, CPU clock speed, network topology, bandwidth, switch delays, cache size, block size and associativity. SPASM has been used to study a range of multiprocessor components and systems, for example, Sivasubramaniam et al studied different interconnection networks and cache sizes [Siv99] and investigated the scalability of shared-memory multiprocessors using a full-map directory protocol with various synchronisation primitives and interconnection networks [Siv94].
The Wisconsin Wind Tunnel system [Rei93], developed by Reinhardt et al at the University of Wisconsin-Madison, executes parallel shared-memory applications on a message passing machine. The application is executed in the processors of the parallel machine (currently a Thinking Machines CM-5), with cache misses being passed to a simulator. The simulator models the cache coherence protocol but not the interconnection network topology or contention; only the interconnection network’s latency is included in the simulation. Large applications can be run in a reasonable time as the application is executed in parallel using the underlying hardware, although in systems that use a large number of processors, the simulation suffers from poor speedup due to the contention for the simulator.

The Wisconsin Wind Tunnel has been used to study a wide range of features of shared-memory multiprocessors. Wood et al [Woo93] compared seven different directory based cache coherence protocols using billions of memory references. Hill et al [Hi193] evaluated a Check-In/Check-Out programming model with a new directory cache coherence protocol using this platform, and compared it to more traditional protocol. Reinhardt et al [Rei96] explored distributed shared-memory on a network of workstations by comparing four alternatives for implementing a cache coherence protocol. The successor, Wisconsin Wind Tunnel II [Muk97] has been used by Lai and Falsafi [Lai99] to investigate a memory sharing predictor to prefetch memory blocks and by Bilir et al [Bi199] to evaluate a multicast snooping protocol on a 32 processor NUMA architecture.

Kubota et al [Kub98] developed a system (EXCITE/INSPIRE) for simulating very-large scale data parallel programs on distributed memory machines. EXCITE is used to annotate the code to produce messages and execution times before it is executed on the host processor. INSPIRE is then used to produce a network simulator that can simulate the messages to produce communication times. The combination of the two times provides information about the whole system. Experiments were performed to assess the impact of cache size, network topology and bandwidth on problems from the NAS parallel benchmark suite [Bai95].
DirectRSIM, developed by Durbhakula et al [Dur99] at Rice University, extended the RSIM application-driven simulator discussed earlier. They improved accuracy and speed when simulating multiprocessors with processors that exploit instruction-level parallelism. DirectRSIM is an execution-driven simulator that consists of two simulation systems. The first is the usual memory system simulator for modelling accesses to memory; the second is the timing simulator that uses a record of the instructions executed to predict the time between memory references. DirectRSIM proved to be more accurate than RSIM and other execution-driven simulators when studying processors exploiting instruction-level parallelism. It also proved faster than RSIM, although slightly slower than simpler execution-driven simulators.

Prylli and Tourancheau [Pry98] proposed a tool that simulates a distributed memory MIMD computer that is capable of running real parallel applications. The simulator executes on a network of workstations or a small parallel machine and is sufficiently parameterised to allow different architectures to be created easily. The parameters that are available include the power of the processor (a single value that represents the processing power of the processors used in the simulated system - it is relative to the processor on which the simulation is executing), the topology of the interconnection network (ring, mesh, hypercube and crossbar), the communication protocol used and interconnection network bandwidth and latency.

Other execution-driven simulations of multiprocessors have been developed to study a wide range of architectural alternatives, a selection of which are outlined. Holt et al [Hol96] investigated speedup, programmability and bottlenecks of distributed shared-memory multiprocessors with 16, 64 and 256 processors, with TangoLite being used to provide the memory references. Dwarkadas et al [Dwa93] studied a variety of release consistency models on a network of workstations connected using Fast Ethernet or ATM with coherency maintained by software. Stenström et al [Ste92] also used Tango (an early version of TangoLite) to compare two types of multiprocessor architecture, with the memory references passed on to a simulation of the memory hierarchy (although only interconnection network delays were modelled). Erlichson et al [Erl94] investigated the effects of clustering in COMA
architectures. The machines consisted of 64 processors with experiments performed for 1, 2, 4 or 8 processor clusters, with TangoLite being used to produce the memory references. Talbot and Kelly [Tal98] used execution-driven simulation to study the performance effects of introducing proxies into cache coherent distributed shared-memory systems. Cox et al [Cox94] modelled two different multiprocessor architectures to compare the performance of hardware and software implementations of shared-memory. Byrd and Flynn [Byr99] modelled a system with 64 processors connected by a 6D binary hypercube to investigate mechanisms to support producer-consumer communication. The model allowed the network dimensionality, bandwidth and latency to be changed easily.

The performance advantage offered by execution-driven simulation has proved very popular when producing simulations of multiprocessor systems, although care must be taken to ensure that the timing between memory references is correct, or incorrect results will be produced. However, execution-driven simulation is not always suitable, in particular when modelling processors that exploit instruction level parallelism, as accurately predicting the time between references is difficult.

Simulation has proved a very popular technique for exploring the many different options available when designing a multiprocessor system. Simulation allows these design alternatives to be evaluated without the expense and long design times associated with hardware construction, and simulation models offer a degree of flexibility that is not available with hardware. This allows changes to be made to the architecture quickly and with a minimum of effort, for example, adding more processors, changing the cache configuration and altering the interconnection network topology.

HASE, the simulation environment developed and used during this project is described in the next chapter. How HASE compares to the simulation environments
outlined earlier is discussed in Section 9.2. A description of the multiprocessor simulation developed is contained in Chapters 5 and 6.
Chapter 4

Hierarchical Computer Architecture
Design and Simulation Environment

This chapter will discuss HASE, the simulation environment that has been used throughout the work carried out. The chapter starts by providing an overview of the history of HASE, from how it started to more recent developments. The numerous features of HASE are also discussed in detail. Parts of the discussion below, detailing the history and features of HASE, have appeared in a paper published recently in ACM TOMACS [Coe98].

4.1 The Evolution of HASE

In 1988 a simulation of the MC88000 microprocessor system on a MIMD transputer network was written as an undergraduate project [Rob91], to investigate the feasibility of producing a general purpose simulator for a transputer network. The simulation was written in Occam2 and included several features which are still central to the HASE system today. Firstly, the entities of the simulation model were represented as different objects, which enabled the model to be distributed easily over the network, with each node executing one object. Secondly, the simulation
was configured to produce a graphical output of the architecture's internal data flows.

The ideas developed in this project, coupled with an increased interest in simulation environments for exploration of computer architecture design trade-offs, saw the start of the development of HASE in 1989. The first HASE system had many components in common with the current HASE system, for example, an architecture description, an architecture animator and an underlying simulation engine. In this early system DEMOS (Discrete Event Modelling on Simula) [Bir85] was used to provide the simulation engine. The graphical front-end was made up of several components, the main ones being a graphical interface for DEMOS, an architecture editor and a trace animator. The custom dialogs and windows were developed using the Motif widget set, based on the X Window System using X11. This HASE system was developed and experimented with by A. R. Robertson as a Ph.D. project at The University of Edinburgh [Rob95].

The next incarnation of HASE system was developed by F. Howell as part of a Ph.D. project [How96a] and featured an entirely X11/Motif interface and Sim++ [Jad91] which replaced DEMOS as the underlying simulation engine. The Sim++ version was implemented to allow HASE to be C++ based, and to take advantage of Sim++'s more advanced features, for example the Time Warped kernel.

The development of HASE continued in 1992 as part of the ALAMO (Algorithms, Architectures and Models of Computation: Simulation Experiments in Parallel Systems Design) [Ibb96a] project. The main change during this period was the introduction of ObjectStore [Obj93] to store architecture designs and entity libraries [Ibb96b]. There were several perceived advantages of introducing the database system into HASE, with the first and main advantage being the ability to persistently maintain architecture design projects and entity libraries. This was an important feature as, before the use of ObjectStore, architecture designs had to be coded in C++ which meant that even the smallest of architectural changes resulted in a recompilation of the project architecture, causing a bottleneck in the design cycle.
Secondly, the transaction processing capabilities of the ObjectStore database system, for example, rollback and nested transactions, provided a solution to version control and facilitated the exploration of alternative design decisions. Thirdly, the database management system allowed multiple sets of experiment results to be stored and maintained, along with the state of the architecture model that produced the results. The addition of ObjectStore also required changes to the user-interface of HASE to allow designs to be created and modified interactively.

From 1995 the continuing development of HASE has been an integral part of the work carried out during this Ph.D. project. Modifications and additions to HASE made during this time are described in the next section, as well as any other relevant developments.

4.2 Recent Developments in HASE

The major changes made to the HASE system during the course of the work described in this thesis are outlined in the sections below.

4.2.1 Project Data Storage

HASE has undergone major changes in recent years. The first of these was the removal of the object-oriented database management system, ObjectStore, as the method of project data storage. Whilst ObjectStore proved satisfactory for a time, it was realised that the licensing restrictions imposed would eventually limit the free distribution of HASE within the academic community. However, this was not the only reason for the demise of ObjectStore in HASE. Several problems were revealed when HASE was used to simulate the H-PRAM model of parallel computation mapped onto large 2-dimensional meshes [Ibb96a]. The most important of these was limited performance, caused by maintaining database integrity whilst interactively manipulating large numbers of entities. Other problems involved the inability to recreate a project if database integrity was breached, the lack of garbage collection
(which caused database sizes to grow rapidly into tens and sometimes hundreds of megabytes) and the problems of allowing multiple users to use the same project database. Most, but not all of the problems occurred because HASE was not designed specifically with ObjectStore in mind, and to solve these would have required a complete redesign of HASE to make the two work in harmony.

To overcome the difficulties experienced with ObjectStore (without reverting back the old method of writing C++ code), an architecture description language was developed. The description language was composed of two files, the Entity Description Language (EDL) file and the Entity Layout (EL) file [Coe97b]. The EDL file describes all of the entities of an architecture (including any associated ports and parameters), how they are connected together and how they fit into the hierarchy, as well as any user defined data types. The EL file includes all of the information needed by HASE to display the architecture design, for example, the coordinates of the entities, where the ports and parameters are to appear and the routing information for connecting links. Figure 4.1 shows an example EDL file and Figure 4.2 shows an example EL file, both for a simple project containing two entities, a sender and a receiver. From now on the term EDL will be used to refer to the name of the language as well as the name of one of the files used. See Appendix A.1 for the EDL grammar.

As can be seen from Figure 4.1 the EDL description of an architecture is split into 5 sections.
1. The preamble section contains general project information, for example, its name, the directory containing all the files and a description of the project. It may also contain a version number and the author’s name.
2. The parameter library section contains the definitions of all the user-defined types used by the project.
3. The globals section declares all of the variables that are global to the project. Any entity in the architecture can access these variables.
4. The entity library defines the entities that can be used to construct the architecture. Two main types of entity defined here are atomic and composite.
Atomic entity definitions contain details of the ports and parameters as well as a textual description. Composite entity definitions contain the same information as atomic entities but also define any composite entity’s children and links present.

5. The architecture section describes the top-level architecture entities and how they are connected together.

PROJECT {
PREAMBLE {
    NAME "Sender-Receiver"
    DIRECTORY "D:\Hase\Projects\SendRec"
    VERSION 1.0
    AUTHOR "Paul Coe"
    DESCRIPTION "Sender and Receiver Project"
}
PARAMLIB {
    STRUCT(DataPkt,[RINT(PacketNo,0),RINT(NoBytes,1)]);
    LINK(SimpleLink,[(DATAPKT,RSTRUCT(DataPkt,DP))]);
}
GLOBALS {
    RINT(PacketsToSend,10);
    RINT(PacketsReceived,0);
}
ENTITYLIB {
    ENTITY Sender {
        DESCRIPTION ("Sender Entity")
        PARAMS(RINT(delay,10));
        PORTS(PORT(Out,SimpleLink,portright));
    };
    ENTITY Receiver {
        DESCRIPTION ("Receiver Entity")
        PARAMS(RINT(delay,10));
        PORTS(PORT(In,SimpleLink,portright));
    };
}
STRUCTURE {
    AENTITY Sender SENDER (DESCRIPTION("Sender") ATTRIB());
    AENTITY Receiver Receiver (DESCRIPTION("RECEIVER") ATTRIB ());
    CLINK(Sender.SENDER[Out]->Receiver.RECEIVER[In],1);
}
}

Figure 4.1: An example EDL file

| SENDER : POSITION (20,20) |
| SENDER : PORT Out SIDE RIGHT POSITION middle |
| RECEIVER : POSITION (100,20) |
| RECEIVER : PORT In SIDE LEFT POSITION middle |

Figure 4.2: An example EL file
There were several advantages to using an architecture description language. Firstly, it overcame the licensing restrictions of ObjectStore, allowing HASE to be distributed more easily. Secondly, architecture designs could now be stored in much smaller and easier to manage text files. A third (and major) advantage of creating the architecture description language was the added flexibility it provided for designers. When ObjectStore was the only method of storing project information, designers were restricted to using the graphical front-end to construct their designs. Although the graphical design method was suitable for initial high-level design work or for beginners, more experienced designers found the user interface to be quite cumbersome particularly for more detailed work. By adding a description language that was read in, or generated by HASE, designers then had the choice of using the graphical interface, or writing descriptions directly with some form of text editor, or working with a combination of the two methods.

4.2.2 Discrete Event Simulation Engine

HASE was originally designed to be used with a commercially available discrete event simulation engine, Sim++, which allowed a working system to be developed more quickly. However, with the removal of ObjectStore, this simulation engine became the only remaining item of licensed software to be used by HASE. With a growing desire to allow people outwith the department to experiment with HASE, the decision was made to write a simulation engine to replace Sim++. This decision was also influenced by the need to run experiments on platforms not supported by Jade, such as Linux and Cray systems. This led to the creation (in 1996) of HASE++ by F. Howell [How96b]. HASE++ is a discrete event simulation engine with very similar functions and data types to those of Sim++, enabling existing project code to be converted easily. HASE++ uses threads, and was implemented as a C++ library that was linked into the simulation at compile-time. This enabled all the standard C++ functions and features to be used when constructing simulations. The thread and synchronisation libraries were the main components that varied from platform to platform; so as long as these libraries existed on the desired platform, the HASE++ simulation library could be ported with a minimum of effort. Another advantage of
having an in-house simulation engine was access to all source code, which allowed additions and modifications to be made quickly and easily.

4.2.3 Modal Operation

The next major change to HASE was the introduction, in 1997, of five modes of operation, namely: (1) Model Design, (2) Model Validation, (3) Build Simulation, (4) Simulate System and (5) Experiment. This facility formalised the architecture design cycle and allowed proper separation of concerns between the different phases of activity. Previous versions of HASE (as well as the current one) relied on pull-down menus to group similar activities together, from which the required action was selected to perform a task. The introduction of modes allowed a more structured menu system to be introduced, with only relevant options for the current mode being made available to the user. HASE uses two menu systems to present the available tools and options. The first, system level menus are accessed from the menu bar at the top of the main window and allow access to the main HASE functions. The second set of menus, entity menus, are accessed by right-clicking on an entity and allow access to HASE functions that affect a single entity.

The restructuring of the menu system was performed with two changes to the design of the interface. The first change involved a reordering of the main menu pull-downs to reflect the different modes. This enabled pull-downs relevant to the current mode of operation to be available, while not allowing the rest of the pull-downs to be selected. Figure 4.3 shows the HASE main menu bar and the mode buttons in two of the five modes of operation (Design and Simulate System). A description of the menu options available in each of the modes can be found in Section 4.3.
The second change involved designing a different set of entity pull-down menus for the different modes. For example, when in Design mode, the entity pull-down menu associated with a memory entity would enable the parameters of the memory to be changed, e.g. number of words, word size, read and write access times, etc. However, when in Simulate System mode, the pull-down menu would allow a file to be loaded into the memory before the simulation starts. Figure 4.4 shows the entity menu pull-down in two different modes (Design and Simulate System) for a memory entity (obtained by right-clicking on a entity).

Figure 4.3: HASE main menu bar in Design and Simulate System modes of operation

Figure 4.4: A memory entity pull-down menu in Design and Simulate System modes
4.2.4 Model Validation and Library Management Tool

The model validation and library management tool (LibTool) was produced by L. Williams, as part of a Ph.D. investigating model abstraction and entity reusability in HASE [Wi199]. A library of entities is described using a language called MEDL (which is based on a subset of EDL). MEDL describes which entities are contained in the library and any associated parameter type definitions. LibTool allows an architecture to be constructed from the library of entities. The types associated with ports that have been connected together can be compared, to check the architecture has been constructed in a correct manner. This architecture can itself be stored in the MEDL file for re-use later. An EDL file can be generated from this architecture model, which can be subsequently loaded into HASE. The tool is also capable of reading in an architecture model in EDL (generated by HASE or by-hand) and validating its correctness.

4.2.5 Microsoft Windows Version of HASE

The most recent developments in HASE have revolved around the creation of a version that executes on a PC running Microsoft Windows (NT or 95). This has been a significant task, requiring a complete rewrite of the interface code as well as major revisions to other pieces of code to improve efficiency. The main motivation behind this was to enable HASE to run on a desktop machine with a widely used operating system, allowing HASE to be distributed more easily. The distribution of HASE and HASE++ is also easier, as only executable files need to be handed out to other users, whereas on Solaris the source code is required to enable the system to be recompiled at the remote site. The move to Microsoft Windows removed the problem of different machines running different, incompatible versions of the same operating system. Consequently both HASE and HASE++ were successfully converted to run under Microsoft Windows. The new interface was developed using a graphical C++ programming environment, which helps to maintain consistency in dialog and menu design. The main improvements made were to the animator,
experiment system and timing diagram. The new system developed is more stable and responsive, as well as executing simulations more quickly.

4.3 The Current HASE System

The current HASE system will now be discussed in more detail. Figure 4.5 shows the software architecture of the HASE system, illustrating the main components of the system and how they interact with each other.

At the most basic level, an architecture design in HASE consists of an EDL file, an EL file and behavioural descriptions written in C++ using the HASE++ library of communication primitives. A hierarchy of entities is described in the EDL file, with each of the entities having a set of ports associated with it. The ports are linked together to form the communication channels for the architecture. Parameter types are also defined in the EDL file, allowing user-defined parameters to be assigned to entities and links. The EL file contains the display information specific to the project. The EDL and EL files can be created in three ways, by hand using an editor, by HASE itself from a design created using the graphical user interface or by using the library and modelling tool. The graphical user interface allows entities and links to be created graphically using drag and drop techniques. When loading a project, HASE reads in the EDL file and creates its internal architecture representation, which it subsequently uses with the information in the EL file to create a graphical representation which is displayed in the design window.

The behavioural descriptions (one for each entity in the design) are combined by HASE as described in the EDL, to create the executable simulation for the architecture. When a simulation is run, HASE takes an input file specifying the parameter settings for the system and creates a trace file detailing the operation of the architecture. This trace file can be subsequently read back in by HASE and used to create visualisations in a number different ways. The experiment control facilities
enable multiple executions of the simulation, creating a set of trace files for a user-defined range of input values for a specified group of architecture parameters.

The restructuring of the menus performed to implement the modal operation resulted in the following main menu options for the different modes:

- In Design mode it is possible to add/remove/copy/paste an entity in the design window, edit an entity’s attributes, create/modify parameter data type definitions, define global parameters and load/save parameter values of all the entities in the project.
• The Model Validation mode allows various checks to be carried out on the design to verify its correctness, for example, whether two entities that are linked together communicate using the same message packet types.

• The Build Simulation mode enables the simulation executable of the created system to be constructed, as well as allowing the level at which the trace file should be generated to be selected.

• In Simulate System mode it is possible to run the simulation and animate the graphical display of the simulation. The user is also allowed to vary system parameters and rerun the simulation.

• Experiment mode enables multiple runs of the simulation to be automatically performed with different parameter settings.

Other more general functions and tools are available in any mode, for example, load/save/print project are available on the File pull-down menu and Timing Diagram, Hierarchy Viewer and Communication Protocol Viewer are available from the Tools pull-down menu.

4.3.1 Internal Architecture Representation

HASE uses four basic components to represent an architecture design: entities, ports, links and parameters.

Entity: An entity is used to represent a component of the architecture. It contains a list of ports, a list of parameters, an icon, a type name, an instance name and a description. The type name / instance name combination for an entity is unique; components may however have the same type name, indicating that there is more
than one instance of the same component. Figure 4.6 illustrates a memory entity in HASE that contains a number of parameters and two ports.

![Image](image_url)

**Figure 4.6: A HASE memory entity**

**Port:** A port is used by an entity to send a message to another entity. By using ports to connect entities together reusability is improved, as a more structured interface is provided.

**Link:** Links represent the communication channels between entities. They connect two entities together by specifying source and destination ports.

**Parameter:** Parameters are used to represent attributes of an entity, for example, a memory entity may have a parameter to represent access time. A selection of parameters for an example memory entity are shown in Figure 4.7.
4.3.2 Model Hierarchy

The desire to support hierarchical models has been one of the major driving forces behind the development of the HASE system. There are two different interpretations of model hierarchy; both are supported to different degrees in HASE. The first is the display hierarchy, which allows the model to be displayed and animated at different levels of the hierarchy, for example, a higher level entity can be expanded to reveal its lower level constituent icons. This feature of HASE has been used effectively in a number of projects, in particular the DASH architecture simulation [Wii95], the Hierarchical PRAM simulation [Ibb96a] and the simulation of a microcontroller [Coe97a]. By allowing more complex parts of a design to be hidden, designers can concentrate on higher level architectural interactions, for example, when simulating the DASH architecture the hierarchical nature of the display allowed low level intranode communications to be hidden, allowing the designer to concentrate on the internode communication protocols. Figure 4.8 shows a multiprocessor with the four
nodes connected to memory via a bus, whereas Figure 4.9 shows the nodes in more detail (with each node consisting of a processor and a cache).

The second hierarchical feature of HASE is hierarchical simulation, i.e., where the simulation can be executed at different levels of the model hierarchy. This allows entities written at a low level (which contain a large amount of simulation detail) to be interchanged in the simulation with entities written at a higher level (which contains less simulation detail but execute more quickly). The hierarchical nature of HASE allows the simulation level to be specified for each entity of the system, which allows a combination of low level and high level entity simulations to be included in a simulation of the complete system. The different simulation levels can be easily changed by toggling a parameter of higher level entities that indicates whether to simulate at this level or at the lower level. This feature also allows the trade-off between simulation speed and accuracy to be explored.
This hierarchical approach to simulation also allows the designer to use well known project building techniques, for example, top down refinement and bottom up construction. The hierarchy viewer shown in Figure 4.10 helps the user ascertain the level of the hierarchy at which the simulation is running. This is extremely useful in large models when different entities are running at different levels in the hierarchy. The viewer shows which entities are executing at which level by surrounding each executing entity in the tree with a double box. Figure 4.10 shows that the model would execute the bus and memory along with four nodes, with two of the nodes being executed at the lower level and two at the higher level.

In practice however, designers have tended not to produce hierarchical simulations, instead opting only to use hierarchical displays. Recently, work carried out by L. Williams [Wil99] as a part of a Ph.D. project attempted to identify the reasons why projects have not been produced at different levels of abstraction and to provide guidelines and methodologies that would enable different levels of abstraction to be incorporated easily into a design.
4.3.5 Architecture Templates

To aid the designer when creating architectures with entities repeated in a regular structure HASE offers architectural templates. These templates provide a method of constructing large architectural structures with a minimum of effort. Simple
templates include a 1D-, 2D-, and 3D-mesh networks and an omega network. These templates are usually parameterised by number of entities in a dimension, number of interconnecting links, whether the links wrap around and the entity to be used for a node in the network. More complex templates also exist, for example, a bus-based shared memory template which is parameterised by the number of nodes, the entities to be used for the nodes, the memories and the bus. This more complex template will be discussed in more detail in Chapter 5.

4.3.4 Visualisation

HASE includes several tools for visualising the output generated by the simulation, the most commonly used being the animator. The animator reads in trace files and uses the information they contain to animate the graphical display of the architecture. The trace files are generated automatically by the simulation, with no need for the user to write explicit animation code. The animation reflects the activity of the simulation in a variety of ways, for example, moving packets between entity icons to represent communications and changing entity icons to reflect a change in an entity’s state. The state of the parameters associated with the entities, including any arrays, can also be visualised during the animation. The main advantage of the animator is that it enables the user to check that the model is performing correctly and producing the correct results. Figure 4.11 shows the animator control panel. The animator provides the user with the ability to rewind, play, step through, stop and pause an animation. The speed at which the design is animated can also be controlled and the time bar enables the user to quickly jump to different parts of the animation.
The results produced by a simulation can also be viewed with the communication protocol viewer, which displays two entities and the events, in chronological order, associated with them. This is especially useful when determining whether two entities are performing correctly according to a predefined protocol. Figure 4.12 shows the communication protocol viewer (with only one of the entities visible). The user can determine which two entities are to be displayed as well as which events are shown, i.e. events generated by entity A, entity B or external entities. Filters can also be applied to the view of events, specifying what message types are to be displayed, and thus allowing the user to concentrate only on events that are of interest.

The timing diagram (Figure 4.13) shows how the states of individual entities and entity parameters vary over the course of a simulation run. There are several views of the data available. The timing diagram is capable of displaying integers and enumerated types. Integer parameters appear as a line graph showing how the value changes over time and enumerated parameters appear as coloured bars indicating the state of the parameter over time. The user can zoom into the display enabling a finer level of detail to be obtained when necessary. Figure 4.13 shows an example of a timing diagram displaying two enumerated parameters (CacheState of the primary and secondary caches) for a specified time period.
Interface Communications Analysis.
Left: layout_model6.caller_inst.modem_inst
Right: layout_model6.pstn_inst

Figure 4.12: Communication protocol viewer
Figure 4.13: Timing diagram

Figure 4.14 presents the percentage time spent in each state of the enumerated parameters shown in Figure 4.13, as a bar and a pie chart.

Figure 4.14: Timing data displayed as percentages
4.3.6 Model Experimentation

The Experiment mode in HASE allows the user to run multiple simulations and control the values of the parameters in each run. The experiment control panel (Figure 4.15) allows the user to select which architecture parameters are to be varied (c.f. Figure 4.7) and to specify the range of values that the parameter can take. HASE will then run all combinations of the specified parameters and store the results. Parameters may be grouped together so that they change value at the same time, so reducing the number of combinations that are run; a group is considered as a single parameter when calculating the combinations to be run. The user specifies an initial value and a final value for each parameter as well as the step to be used. The step can include simple expressions, for example *2 for multiplication by two at each stage. Figure 4.15 shows the experiment control panel with a selection of parameters, in which two (BlockSize of primary and secondary caches) are grouped together.

The features of HASE described in this chapter combine to produce a system with the flexibility to be used for a wide range of tasks, ranging from teaching applications [Coe96, Coe97c] to experimental architecture design. Using the HASE system to design parallel architectures is the main focus of this work, and this will be discussed in depth in the next chapter. The extensions to HASE to support the model are discussed in Chapter 7.
Chapter 5

Design of the Multiprocessor Model

This chapter outlines the need for a multiprocessor model that enables different design alternatives to be evaluated with a minimum of effort. This is followed by a detailed discussion of the multiprocessor model designed and the parameters that each entity (HASE terminology for a system component) supports.

5.1 Motivation

In Chapter 3 several different approaches to multiprocessor simulation were outlined, along with numerous examples of the multiprocessor simulations that have been developed.

Before discussing simulation and its uses in multiprocessor evaluation, analytical models should be mentioned. In Section 3.2 analytical models were discussed as an option when evaluating multiprocessor systems; however, the work carried out and described in this thesis uses simulation as an evaluation tool, not analytical models. This is due to the need to evaluate complete multiprocessor systems in enough detail to allow decisions about the performance to be made with a reasonable level of confidence and to enable different architectural designs to be explored. To
accomplish this with analytical models is extremely difficult, as the complex nature of multiprocessor systems, combined with the flexibility required to explore different architectures cannot easily be expressed with analytical techniques. The model would also need to be capable of dealing with real applications and benchmarks, something not ideally suited to analytical modelling.

The multiprocessor simulations that have been developed [Hil93, Dwa93 and Arc86] (either using a simulation environment or framework, or by writing one using only a simulation language) are created to evaluate possible multiprocessor design alternatives. They are used to study the impact of a new solution or approach to a particular area of multiprocessor systems, to compare previously presented solutions or to assess the performance of a proposed design for a multiprocessor machine.

When studying new solutions or approaches to architectural problems (for example a new cache coherence protocol or a new solution to the memory consistency problem) these need to be compared to existing solutions to show that they offer an advantage. This usually means demonstrating that the new solution outperforms the old one(s). Simulation has proved a popular technique through which to show the possible advantages of a new solution. Eggers and Katz [Egg89] presented a new cache coherence protocol to overcome the shortcomings of the more traditional Firefly and Berkeley protocols. Hill et al [Hil93] showed how a Check-In/Check-Out model of programming, using a specific protocol, could offer a performance advantage over more traditional directory protocols.

Other research has focussed on comparing solutions and approaches that have been proposed by other people, in an attempt to allow informed decisions to be made when designing a multiprocessor and rule out alternatives that are not viable options. Dwarkadas et al [Dwa93] compared several implementations of lazy release memory consistency models, Wood et al [Woo93] used simulation to compare seven different directory based protocols, and Archibald and Baer [Arc86] compared the performance of eight snoopy cache coherence protocols.
Many simulations of specific multiprocessor systems have also been constructed to enable an evaluation of the proposed architecture to be performed before construction. Sequent Computer Systems built a simulator of the STiNG multiprocessor [Lov96] before construction and Chaiken et al [Cha91] produced a complete simulation of the Alewife system to assess the performance of the cache coherence protocol to be used.

Almost all of the simulations developed allow a small number of architectural parameters to be changed, in addition to the changes that can be made to evaluate the component being investigated. The most common parameters that can be changed easily are the number of processors and the cache size, and simulations that focus on interconnection networks often include latency and bandwidth parameters.

The problem with only being able to change a small, limited number of architectural parameters is that the impact of a new solution cannot be evaluated fully. For example, if a new cache coherence protocol is implemented on a clustered distributed shared-memory machine, the organisation of the processors and clusters will affect the performance, but may not be the only factor. The way in which synchronisation is implemented may have a serious bearing on performance, as could the interconnection mechanism used. It is very difficult to evaluate an architectural component by changing only a limited number of parameters, as all the different components interact in ways that are not always easy to predict. By allowing more architectural components to be changed, it is possible to perform a more complete evaluation of a new solution or technique. For example, if a new cache coherence protocol is implemented and does not perform as well as expected, a change in a different area, for example the implementation of the synchronisation primitives, may improve the performance.

The multiprocessor simulations developed and outlined earlier (see Section 3.3) do not easily allow many aspects of the architecture to be changed. None of the simulations presented allows the designer complete freedom to change, with a minimum of effort, any part of the multiprocessor architecture. It may be necessary
to change the design of the cache, include a different cache coherence protocol, change the interconnection network or simply add more processors or clusters of processors. The ability to do this would allow the designer to explore and evaluate fully a multiprocessor architecture.

The project aimed to develop a simulation model and framework that would enable multiprocessor architectures to be changed relatively easily, allowing designers to explore different architectures and experiment with new designs.

The remainder of this chapter deals with the design of the multiprocessor simulation model.

5.2 Design of the Multiprocessor Architecture Model

As discussed in Chapter 2, there are many different types of multiprocessor architecture and two main models of communication (message passing and shared-memory). As well as these different multiprocessor systems there are also many decisions that need to be made regarding the cache coherence protocols, memory consistency models, synchronisation primitives and interconnection networks to be used. This section describes the architecture to be modelled and all the different architectural features to be included.

The model created concentrates on shared-memory multiprocessors that fit into the UMA or NUMA type of machine. Message passing machines and COMA shared-memory multiprocessors are not implemented, although Section 9.3 discusses how the model and framework could be extended to include these types of machine.

It is much easier to develop applications for shared-memory multiprocessors than for message passing multiprocessors due to their simpler programming model; however the lack of scalability of shared common memory systems has meant that they have had limited success, both in research and commercially. However, by using the
distributed shared-memory approach the scalability is improved (allowing machines with larger numbers of processors to be constructed), whilst maintaining the ease of programming of traditional shared-memory multiprocessors. Distributed shared-memory machines have therefore resulted in an increase in the number of multiprocessors being constructed, with Silicon Graphics Incorporated, Sun Microsystems and Sequent all offering a range of distributed shared-memory machines.

However, distributed shared-memory multiprocessors have limitations of their own. The use of more scalable interconnects (not buses) results in inefficient broadcast mechanisms which means that snoopy protocols are no longer suitable for maintaining coherence. Consequently, coherence actions take longer, which might decrease system performance. To try to construct a shared-memory system that is scalable and maintains high levels of performance, processors are being grouped together to form a cluster which share a local memory, these clusters are then joined together to form larger systems. A typical clustered distributed shared-memory multiprocessor is illustrated in Figure 5.1. This idea can be taken further; by considering the complete distributed multiprocessor systems as clusters in a larger system, more hierarchical levels can be added to the architecture. The work carried out during this project concentrates on multiprocessors that contain simple shared-memory multiprocessors in a cluster and not the hierarchical structure. This simplifies the design and implementation, although Section 5.2.8 discusses how the framework and model could be extended to include multiprocessors with a more hierarchical nature.

The design of the system is discussed from the bottom up, starting with the individual components, followed by the clusters and then the whole system. The discussion of the design contains a description of the function of the components, the parameters that can be changed to alter the behaviour of each entity and how each entity communicates with other entities connected to it.
5.2.1 Processor Entity

The processor entity is responsible for driving the simulation; it issues memory requests (both reads and writes) and deals with data returned by the memory system. The memory requests could be for instructions, data or synchronisation. There are several ways in which these memory requests can be generated, distribution-driven, trace-driven, application-driven, and execution-driven; these have been outlined in Section 3.3.2.

Three of these approaches (trace-driven, application-driven and execution-driven) are feasible for the simulation of the distributed shared-memory system being developed. Figure 5.2 shows the three different types of processor entity and examples of how the application or benchmark drives them.

In a trace-driven processor, Figure 5.2(a), a file containing the sequence of addresses is processed by the processor entity. The trace file informs the processor whether to perform a read or a write, to which address to issue the request and at what time to issue the request. The application-driven processor, Figure 5.2(b), has an associated file that contains the assembly code instructions to be executed. The simulated
processor executes the instructions and the processor generates the memory requests when new instructions or data are required, and also when synchronisation points are reached. Figure 5.2(c) shows the execution-driven processor. Here the processor actually executes the high-level application code on the host machine as part of the behavioural code of the processor entity. When new data is required, or a synchronisation point is reached, a memory request is issued.

The type of processor entity used influences the parameters associated with it. For trace-driven processors there are very few parameters that can be included to influence the execution of the trace. The main parameter is a scaling factor that would affect the time recorded in the trace file between the events. In an application-driven processor the behaviour of the processor is modelled in sufficient detail to allow the application or benchmark to be executed on it. By simulating at this level of detail, the processor is able to contain parameters that would enable the execution times of the different instructions to be altered. As with application-driven
processors, execution-driven processors also need to know how long different types of instructions take to enable them to estimate the time between successive memory requests.

Trace-driven processors are simple to implement once the design of the trace file has been decided upon. However, the ability to evaluate different processors accurately is limited, as the trace file may only contain the time between successive memory requests; the instructions that were executed between these requests are not recorded, making it difficult to estimate how a different processor would execute the trace file. The only adjustment that can be made is to scale the values between all of the requests, effectively modelling a faster or slower processor. The other problem with a trace-driven processor is that no computation is performed upon the values returned by the memory system, making it difficult to ensure that the system is performing correctly. To enable computation to be performed the trace file would have to contain a record of the instructions executed that operate on the data returned, but this would effectively make this an application-driven processor as the trace file would, in effect, specify a sequence of instructions for a simple RISC processor.

The second type of processor, application-driven, is complicated to design and implement, the main reason being that the processor entity must be capable of accurately executing all of the instructions used by the applications or benchmarks to be executed. There are several approaches that have been taken to produce accurate processor models. The processor models in SimOS [Ros95] use binary translation (see Section 3.3.1), the CacheMire Test Bench [Bro93] uses instruction set emulation (see Section 3.3.2(iii)) and RSIM [Pai96] (see Section 3.3.2(iii)) simulates the components of a processor, enabling the instructions to be executed. The different approaches have to be designed very carefully so as not to significantly slow down the simulation execution. However, the faster the processor simulation is made, the less information can be extracted and the fewer parameters can be included. For a detailed processor simulation, individual components can be parameterised, allowing very fine control over the attributes of the processor; however this level of detail will drastically increase the simulation execution time. For a processor component that
models instructions at a higher level, parameters can be included for each instruction to allow different processor configurations to be evaluated. Although application-driven processors allow fine control over the processor configuration and allow accurate results to be obtained, they are difficult to implement efficiently as they execute every instruction in the application to be executed.

The final type of processor illustrated, execution-driven, executes most of the instructions of the application on the host machine. The application is coded into the processor entity and compiled into the simulation. When new data is required or results are to be written back, calls to the memory system simulation are used (not loads and stores on the host machine). This retrieves the data from the simulated memory and not the physical memory of the host machine. The memory system simulation is synchronised with the application by inserting code to stall the processor component before each memory request. The length of the stall is determined by the instructions that were executed since the last memory request. The length of stall is the estimated time it would have taken to execute these instructions on the simulated processor. This type of processor entity executes the application more quickly than an application-driven processor, as most of the instructions are not simulated. However, the accuracy of the results is slightly compromised as an estimated time between memory requests is used rather than the actual time used in application-driven processors. Execution-driven processors can contain parameters that specify the length of time different instructions take to execute, allowing the time between memory requests to be adjusted, enabling different processors to be evaluated.

Based on these pros and cons, the processor entity used in the multiprocessor model created was chosen to be execution-driven. The ability to adjust the execution time of individual instruction execution times provides enough flexibility to evaluate different processor configurations. Execution-driven processors are also much easier to implement efficiently and provide better performance than most application-driven processors. The application-driven processor implementations that are almost as fast, for example those implemented using binary translation techniques, do not offer
enough flexibility, and trace-driven processors do not provide enough flexibility to model different processors. The other type of simulation mentioned in Section 3.3.2, distribution-driven, is not considered as it is not possible to evaluate the performance of the multiprocessor system accurately using applications or benchmarks.

One assumption that has been made when designing the processor is that only data accesses to the memory system are being modelled. Instructions retrieved from private memory are usually fetched via a separate instruction cache and are read-only and so have little impact on the performance of the shared-memory part of the system, apart from using part of the available local network bandwidth. All three possible processors support this assumption, and all three, if required, can be modified to fetch their instructions from the shared-memory, or from a separate private local memory.

To better understand the delay (and other time related) parameters for the entities described, it should be stated that the value of the delay is not in real time. The value represents the number of simulation time units, which is an abstract value with no relation to physical time. The values defined by delay parameters are all relative to each other, for example, an integer add instruction may be assigned a delay of 1 time unit and an integer multiply instruction a delay of 5, meaning that the multiply instruction will take five times longer than the add. Similarly, if a cache entity is given an access time of 1 time unit, the length of time taken to access the cache will be the same as the time to perform an integer addition. Different systems can therefore be compared using their simulation time (this is not to be confused with the simulation execution time). The simulation time of the system or individual component can be converted to a real time by assigning a real time value (for example 10 ns) to 1 unit of simulation time.

An execution-driven processor entity requires parameters that specify the execution times of the different instructions. However, there are other features of the processor that affect the operation and performance of the system, for example, the synchronisation primitives and memory consistency model.
Almost all parallel applications require synchronisation primitives to ensure correct operation. The processor must supply these primitives to the application, and the underlying implementation can have a significant impact on system performance. The ability to change easily the implementation of these primitives would allow different implementations to be evaluated with a minimum of effort. The processor entity therefore needs a parameter that specifies the synchronisation primitives to be used and their implementation.

The memory consistency model used by the system specifies which instructions are allowed to be outstanding (those that are issued but not yet completed) when a new instruction is issued. It is the memory system that must change most to support different consistency models (for example, caches that support non-blocking loads); however the processor must be aware of how many instructions are outstanding, what type they are (reads, writes or synchronisation accesses) and whether it is allowed to issue the next read, write or synchronisation access. The processor entity therefore requires a parameter that indicates the memory consistency model to be used; this parameter controls the issue of instructions from the processor.

Another factor that affects the performance of the system is the amount of data that can be transferred between the processor and the memory system, especially in applications that operate on double precision floating point numbers. By increasing the communication bandwidth between the processor and the memory system, a 64-bit number can be fetched in a single cycle, reducing the execution time. By comparing the performance to a 32-bit bandwidth connection, the increase in performance can be evaluated to decide if the extra expense of a wider connection is worthwhile.

Table 5.1 summarises the parameters of the processor entity that have been discussed in this section. Section 6.2 describes the implementation of the processor entity and its parameters.
<table>
<thead>
<tr>
<th>Processor Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Width</td>
<td>The width of the connection to the memory system</td>
</tr>
<tr>
<td>Synchronisation</td>
<td>Used to specify the type and implementation of the synchronisation primitives to be provided</td>
</tr>
<tr>
<td>Memory Consistency Model</td>
<td>The consistency model to be used by the processor</td>
</tr>
<tr>
<td>Instruction delay</td>
<td>The processor requires a delay parameter for each of its instructions; the delay specifies the execution time for the instruction</td>
</tr>
</tbody>
</table>

Table 5.1: Parameters of the processor entity

The final area of design is the interface of the processor with the memory system. The processor needs to send read, write and synchronisation requests to the memory system; it also needs to receive data and acknowledgements from the memory system. The acknowledgements are needed to inform the processor that earlier requests are completed, enabling the memory consistency model to operate correctly. An example of why acknowledgements are needed is that a write to an address does not return any data, but a processor using a sequential consistency memory model must know that the write has completed before issuing the next request. The only way the processor can know that a request has completed is through the receipt of an acknowledgement message. Section 6.1 details the implementation of the message types used by the processor to communicate.

This completes the design of the processor entity used to drive the multiprocessor simulation.
Caches are a well understood component of modern computer systems and have been studied at length. However, there are still many areas of cache behaviour (especially within multiprocessor systems) that need to be explored. The function of the cache is to service as many memory requests issued by the processor as possible, without involving components lower in the memory hierarchy. This means that the cache has a significant impact on the performance of the system and care must therefore be taken in its design. The cache entity included in the simulation must enable the designer to explore the different configurations of the cache to allow the most suitable one to be chosen.

Unlike the design of the processor entity, which had several possible types, the cache entity has only one possible type. The cache receives requests, looks to see if these requests can be satisfied by the data held locally and, if not, passes them on to the next level and waits for the response of the lower level entity. The main design decisions here revolve around what the cache should contain and what parameters should be included.

The first area to consider is the content of the cache, i.e. what each line of the cache contains and the total number of lines. A cache line contains the information retrieved from memory; it also needs a tag that uniquely identifies the data stored. The final contents of a cache line are a small number of status bits. Although these general fields are reasonably standard, the required number of status bits and their use varies from cache to cache, as does the amount of information stored per line. The cache entity must be capable of dealing with these differences. Figure 5.3 shows the structure used by the entity to represent a cache line.

![Figure 5.3: The structure of a cache line](image-url)
Almost all caches include a valid bit (used to indicate whether the rest of the cache line contains correct data), so a separate valid bit was included. The other option was to use one of the status bits, but as almost all caches have some form of valid bit, a separate valid bit would make implementation easier and the valid cache lines would be easier to spot when displayed in HASE.

The second part of the structure is the tag. This is a portion of the address that uniquely identifies the data stored in the cache. It is used when searching the cache and when constructing the memory address to which to write the data when it is removed from the cache.

The next part of the cache line, the data, is of a variable size to allow cache configurations with different amounts of data per line to be evaluated. When data is fetched into the cache, data from neighbouring memory locations are also fetched to fill up the cache line. As most programs exhibit a certain degree of locality, the data required by future accesses may already be in the cache. This prefetching can significantly improve performance. However, if the cache lines are too large, the time taken to fetch all the data from memory may negate any prefetching advantage, as much of the data will not be accessed. A further complication of large cache lines in shared-memory systems is the problem of false sharing. This is where two caches of different processors contain the same data (referred to as sharing); however they are not accessing the same locations, i.e. one processor may only be interested in the lower half of the cache line whereas the other processor may only be interested in the upper half. Shared data can be a major source of performance degradation in shared-memory systems; false sharing should therefore be avoided as much as possible. Consequently the size of the cache line is an important consideration when designing a cache and the ability to change its size easily in a simulation allows design trade-offs to be examined. The size of the cache line data is represented in the cache entity by the block size parameter.
The final section of the cache line structure is the status bits, and here there are two options. The first is to provide a variable number of bits controlled by a number, as in the cache line data. The second is to use a single integer to represent the state. Although the first option seems to be the best solution, the implementation becomes more complicated and the extra work involved in decoding the status would slow the cache down. This approach was necessary for the data, but for the status a single integer value is sufficient. It provides the designer with up to 32 status bits and can be queried and manipulated quickly.

Once the structure of the cache line has been decided, the next area to consider is the size of the cache. There is no fixed size for a cache. Caches closer to the processor are usually smaller than those further away as they are made from faster, more expensive parts. The size of the cache included in the entity must therefore be variable; allowing different configurations to be used both in different experiments on the same system and in different caches of the same system. The cache entity designed here uses the number of lines to specify the size of the cache. The actual storage capacity of the cache is obtained by multiplying the block size by the number of lines.

After the physical structure of the cache has been determined, there are still several questions that need to be addressed in order to make a cache perform effectively. These include:

- Where in the cache should the information be placed?
- How is the information located?
- What happens when data is written to the cache?

The parameters provided by the cache entity should enable the designer to fully explore each of these questions. The cache entity contains parameters for commonly used cache design alternatives which allow the most effective solutions to these questions to be found, for example, associativity, write policy, allocation policy and replacement policy. The associativity parameter allows the cache to be either direct-mapped, fully associative or set associative (of any set size). The write policy
specifies whether the cache is a write-through or copy-back cache. The allocation policy determines whether the data is fetched into the cache when it cannot satisfy a write request. The replacement policy determines how the cache line to be replaced is decided upon.

There are other factors of the cache design that will impact on the overall performance of the system, for example, access time, bus widths and coherence protocol. The cache entity provided must also allow the designer to control these factors.

Parameters for the physical delays associated with accessing the cache are included in the entity. The delays associated with read and write accesses are included separately. The delay parameters are crucial as they allow different speeds of caches to be included in the design. Faster caches are much more expensive, so the performance advantages of these components must be evaluated to assess the possible advantages of spending extra money. The delay parameters also allow caches to be included in a hierarchy, with faster ones nearer the processor and slower ones nearer the memory.

The amount of data the cache can send and receive will have an impact on the system performance. Ideally the cache should be capable of accepting all the information required for a cache line in one cycle; however this may be expensive, so the ability to change the amount of data transmitted/received in one cycle is important. This enables the designer to assess the impact of narrower or wider bus designs on the performance of the whole system. There are two buses that connect the cache entity to the rest of the system, one going to entities higher in the memory hierarchy and one going to entities that are lower. These two connections do not have to be the same width; this means that two bus width parameters are required.

The final design parameter included in the cache entity is the cache coherence protocol. The cache coherence protocol is an important part of any shared-memory system and will have a significant impact on the cache and the whole system, so the
ability to change between different protocols easily is extremely important. The cache coherence protocol used can be complicated and the parameterisation of the cache to allow these to be changed easily is not straightforward. The other parameters discussed for the cache are for the most part numerical and with careful design and implementation can be included into the entity's behaviour without too much effort. The protocol however, contains a significant amount of code with a more complicated interaction with the cache requiring a different mechanism to allow these to be changed by a single parameter. This process is discussed in more detail in Chapters 6 and 7, which detail the implementations of the entities and the extensions to HASE that were required to support these parameters. The synchronisation and memory consistency model parameters of the processor entity also require a more complicated mechanism and are implemented using the same mechanisms developed for the cache coherence protocol. The different cache protocols implemented, how they interact with the cache and how others can be added are discussed in more detail in Section 6.3.1.

Table 5.2 summarises the parameters of the cache entity that have been discussed in this section. Section 6.3 describes the implementation of the cache entity and its parameters.

The final design decisions regarding the cache entity involve the communication interface. The cache is required to communicate with entities both higher and lower in the hierarchy, and communication ports can be added to allow the cache to issue and receive requests and data in both directions. Section 6.1 details the implementation of the message types used by the cache to communicate.

5.2.3 Memory Entity

The next entity to be considered is the memory. It is responsible for storing the information that is used by the application. It must be capable of receiving read, write and copy-back requests as well as supplying data and acknowledgements of actions that have completed.
<table>
<thead>
<tr>
<th>Cache Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Up Width</td>
<td>The width of the connection up the memory hierarchy</td>
</tr>
<tr>
<td>Bus Down Width</td>
<td>The width of the connection down the memory hierarchy</td>
</tr>
<tr>
<td>Block Size</td>
<td>The number of data items that can be stored in a cache line</td>
</tr>
<tr>
<td>Cache Lines</td>
<td>The total number of lines in the cache</td>
</tr>
<tr>
<td>Allocation Policy</td>
<td>Determines whether to fetch the data into the cache on a write miss</td>
</tr>
<tr>
<td>Write Policy</td>
<td>Determines what actions to perform when a write is received</td>
</tr>
<tr>
<td>Replacement Policy</td>
<td>Indicates how to select the cache line to be overwritten when the cache is full</td>
</tr>
<tr>
<td>Coherence Protocol</td>
<td>The protocol to be used to ensure the cache contents are coherent</td>
</tr>
<tr>
<td>Associativity</td>
<td>Determines how the cache is divided up</td>
</tr>
<tr>
<td>Read Delay</td>
<td>The delay associated with a read request</td>
</tr>
<tr>
<td>Write Delay</td>
<td>The delay associated with a write request</td>
</tr>
</tbody>
</table>

Table 5.2: Parameters of the cache entity

Within this simulation the memory entity is used to store only the data used by the applications and not the instructions. This was discussed in Section 5.2.1, where it was stated that the instructions would be stored in a private memory or file associated with each processor.

The memory is a relatively simple entity with a limited number of parameters. As with the cache, the first decision involves the contents of the memory file, in particular what structure it should have and how big it should be. Unlike the contents of the cache, the contents of each piece of memory are much simpler, with an integer being used to represent each word. The number of words in memory is controlled by a parameter to allow different sizes of memory to be investigated.
The memory needs to know how much information to transfer to the next level up in the memory hierarchy. In simple systems this will be the same as the amount of data that fits in a cache line. However, it is possible to design systems that have different amounts of data that are transferred between different entities of the memory hierarchy. Therefore to provide enough flexibility to define the amount of data to be returned by the memory entity, a block size parameter is added.

The next parameter of the memory entity that affects the performance of the system is the time it takes to access and update the data stored. As with the cache entity, the memory entity includes parameters that allow the designer to modify the length of time it takes to perform a read or write request.

The final parameters of the memory entity are the communication bus width and the coherence protocol. The communication bus width specifies the width of the bus that connects the memory entity to the rest of the memory hierarchy. The coherence protocol specifies the protocol used to maintain a coherent system (for more details on the coherence protocol parameter see Section 6.3.1).

Table 5.3 summarises the parameters of the memory entity that have been discussed in this section. Section 6.4 describes the implementation of the memory entity and its parameters.

The memory entity also requires communication ports to allow requests and data to be received and data to be sent out to satisfy a request. Section 6.1 details the implementation of the message types used by the memory to communicate with other entities.
<table>
<thead>
<tr>
<th>Memory Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Width</td>
<td>The width of the connection to the memory hierarchy</td>
</tr>
<tr>
<td>Memory Size</td>
<td>The number of words of storage in the memory</td>
</tr>
<tr>
<td>Block Size</td>
<td>The amount of data that should be returned on a request</td>
</tr>
<tr>
<td>Coherence Protocol</td>
<td>The protocol to be used to maintain a coherent system</td>
</tr>
<tr>
<td>Read Delay</td>
<td>The delay associated with a read request</td>
</tr>
<tr>
<td>Write Delay</td>
<td>The delay associated with a write request</td>
</tr>
</tbody>
</table>

Table 5.3: Parameters of the memory entity

5.2.4 The Basic System

Although the entities have been designed for a multiprocessor system, the three entities that have been described in the previous sections are sufficient to construct a simple uniprocessor system. A processor can be connected to a memory via any number of caches (see Figure 5.4).

![Figure 5.4: A uniprocessor system](image)

This arrangement allows the memory system of a uniprocessor to be evaluated and explored. The designer could run applications or benchmarks through a proposed memory system that could contain an arbitrary number of levels of caches, each with a different parameter configuration. Section 8.2 demonstrates some sample experiments that could be performed using this system configuration.
Although this type of system can prove valuable for evaluating different cache configurations, the aim of this work is to provide a framework for modelling multiprocessor systems. Therefore this basic system needs to be extended to allow multiple processors to be included within the same system. To allow for this, components are needed to link multiple processors together to allow them to access single or multiple memory entities.

5.2.5 Bus Entity

The simplest method of connecting multiple processors to a single memory is to use a bus. A bus shares the access to the memory entity between the processors by only allowing one processor to have control of the bus at any one time. The bus is responsible for forwarding all requests to memory and returning any data from memory to the processors.

The bus is a simple component with only a few characteristics that have an impact on its performance, resulting in few parameters. The most important of these are the bus width and the cycle time, i.e., the time it takes for the bus to receive a request and pass it on the appropriate component. The ability to vary the bus cycle time in a simulation is important as the speed of the bus could have a serious impact on the performance of the overall system. This is because in a system where multiple processors use the bus to access a single shared resource it could be an area of congestion. To enable a designer to experiment with different bus speeds, a bus cycle time parameter is included in the entity.

The bus width affects the number of cycles required to transfer all the data for a particular request. An obvious width for the bus is to make it the same as the number of words of data stored in a cache line. Making it wider than a cache line would probably result in wasted resources as it is unlikely that these extra words would ever be used, unless a more complicated bus was designed that could service multiple request in the same cycle. However, for large cache lines this may prove too
expensive and so the ability to explore different widths of bus is important. A parameter is provided for the bus entity that allows the designer to change the width of the bus easily.

When implementing a snoopy based cache coherence protocol, the bus is an important component and must therefore be aware of the protocol being used. The protocol must also be changed easily in order to see how the system performs with a different protocol. As with the cache and memory entities, a parameter is provided that allows the cache coherence protocol to be changed (for more details on the coherence protocol parameter see Section 6.3.1).

The next parameter determines the number of nodes that are attached to the bus. A node may contain a variety of different structures, for example a single processor, a processor with a number of levels of cache, or even a small multiprocessor. The bus is unaware of the contents of the node; it receives requests and data and forwards them to the appropriate place (which depends on the protocol being used). The structure of the node does not affect the basic behaviour of the bus, only the protocol that the bus is using. The ability to control the number of nodes on the bus is important. A seemingly obvious way to improve system performance is to increase the number of processors; this may be fine up to a point, but eventually there will be too many nodes for the bus to handle efficiently. The bus can only deal with so much information in each cycle and if many nodes are requesting the use of the bus, the amount of time that some nodes will have to wait for control of the bus will increase. This results in processors being stalled for longer and the whole system slowing down. The designer must therefore be able to evaluate how the system performs with different numbers of nodes connected to the bus; allowing the point at which the number becomes detrimental to system performance to be determined.

The final parameter of the bus affects how the bus determines which entity is allowed to use the bus next. There are many different schemes (usually referred to as bus arbitration schemes) that could be used, for example, first come first served, round robin and priority queue. Each of these would result in a different bus control
pattern, for example, priority based approaches could result in some entities having much more bus time than others, whereas the round robin approach ensures that all processors get their fair share of the bus. Different systems or applications may benefit from different bus arbitration schemes, so a parameter that specifies the scheme to be used is provided.

Table 5.4 summarises the parameters of the bus entity that have been discussed in this section. Section 6.5 describes the implementation of the bus entity and its parameters.

<table>
<thead>
<tr>
<th>Bus Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Width</td>
<td>The width of the connection to the memory hierarchy</td>
</tr>
<tr>
<td>Bus Cycle</td>
<td>The time it takes the bus to perform an action, either forwarding a request or returning data</td>
</tr>
<tr>
<td>Coherence Protocol</td>
<td>The protocol to be used to maintain a coherent system</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>Controls the number of nodes attached to the bus</td>
</tr>
<tr>
<td>Arbitration Scheme</td>
<td>The scheme used to determine which node gets control of the bus next</td>
</tr>
</tbody>
</table>

Table 5.4: Parameters of the bus entity

The bus entity also requires communication ports to allow requests and data to be sent to the appropriate entity. The bus entity is different from the other entities discussed so far in that it does not have any defined ports. This is because the number of ports is dependent on the number of processors connected to it. The ports therefore need to be created when the architecture is created in HASE, based on the number of nodes parameter. This is outlined in Section 6.7, which details the implementation of a small-scale multiprocessor system.
A shared-memory multiprocessor can now be constructed from the four different architecture entities that have been described in the previous sections. The bus entity enables multiple nodes to be connected to a common memory entity. Figure 5.5 illustrates a typical shared-memory multiprocessor architecture that can be constructed from the four components.

The nodes shown can contain any number of levels of cache and caches can also be placed between the memory and bus. This allows the impact of cache performance on both sides of the bus to be evaluated. For example, by placing caches on the memory side of the bus, and not on the processor side, the need for cache coherence is removed, so simplifying the system. However, the problem with this system would be increased bus traffic, since many requests can be satisfied by caches on the processor side of the bus without involving the bus. From a simulation perspective, allowing the designer to place any number of levels of caches on both sides of the bus, different architecture configurations that vary in complexity and performance can be evaluated.

The simulation entity used to represent the structure of the shared-memory multiprocessor also has a variety of parameters. These enable fundamental aspects of the multiprocessor architecture to be changed easily. The parameters of this entity control how the system is constructed and what entities are used to construct it. There are three parameters (node entity, network entity and memory system entity) that specify the entities to be used. Each of these entities consists of one or more sub-entities enabling more complicated systems to be constructed, for example, the node entity could be constructed from a processor and two cache entities. The other parameter of this shared-memory multiprocessor entity determines the number of nodes to be connected to the bus.
Table 5.5 summarises the parameters of the multiprocessor entity that have been discussed in this section.

This system can be used to study many aspects of multiprocessor architecture. The model created allows different cache configurations to be evaluated as well as the impact of adding more levels of cache, on both sides of the bus. The processor...
parameters allow different processor speeds to be tried in the multiprocessor, as well as different memory consistency models. The other parameters that have been included in the model enable the designer to assess the impact on performance of a variety of other important architectural features including cache coherence protocols, bus arbitration schemes, cache and memory speeds and different bus widths. Section 8.3 describes some experiments that have been performed using this shared-memory multiprocessor model.

<table>
<thead>
<tr>
<th>Multiprocessor Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Entity</td>
<td>Specifies the entity to be used for the node</td>
</tr>
<tr>
<td>Network Entity</td>
<td>Specifies the entity to be used for the network</td>
</tr>
<tr>
<td>Memory Entity</td>
<td>Specifies the entity to be used for the memory system</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>Controls the number of nodes attached to the network</td>
</tr>
</tbody>
</table>

**Table 5.5:** Parameters of the multiprocessor entity

Although this multiprocessor model enables the designer to evaluate the performance of a large range of systems, it does not allow systems that use distributed or multiple memory entities to be evaluated. The following sections discuss the entities required to support the investigation of this type of architecture.

### 5.2.7 Multiple Common Memory Entities

In the previous section the model of a shared-memory multiprocessor used a single memory entity that was equally accessible to all of the processors in the system. This arrangement is simple to implement in practice and has proved popular for small-scale commercially available systems. The main reason for this is the use of a
shared bus, which allows all of the requests to be broadcast efficiently to all of the
nodes, enabling snoopy cache coherence protocols to be employed.

However this type of system has some limitations. Firstly, the shared bus makes
scaling to large numbers of processors impossible because, apart from the electrical
and physical constraints, contention for this shared resource would drastically reduce
performance. Even using some form of pipelined bus that would effectively allow
multiple nodes to have access to the bus would not solve the problem; the point of
contention would move to the single memory entity. To overcome this, systems
must be allowed to contain more than one memory entity. These can be incorporated
into the system in one of two ways.

The first is to connect the nodes to separate memory entities with an interconnection
network, enabling each of the processors equal access to each of the memory entities
(illustrated in Figure 5.6).

![Figure 5.6: Multiple common memory multiprocessor](image)

The network is no longer limited to a bus; other forms of network can now be used,
for example, multistage networks. Using these other forms of network usually
requires the number of memory entities be the same as the number of nodes. This
reduces the contention for the shared resources of the simple multiprocessor described in Section 5.2.6. Snoopy cache coherence protocols are no longer appropriate, however, requiring directory protocols or other solutions to be used.

This approach solves the high levels of contention for the network and memory entities when large numbers of nodes are used. However, it does not reduce the time taken to request private data or data accessed by only one processor, as requests for data that is not cached still have to traverse the network and be checked by the coherence protocol. Using a directory protocol would reduce the amount of traffic and processing, as the request would not have to be sent to all the caches in the system as with snoopy protocols.

A similar simulation entity to the one used to represent the simple shared-memory multiprocessor has been created to represent this form of multiprocessor. The entity has five parameters, the node entity, the network entity, the memory system entity, the number of nodes and the number of memory entities. Table 5.6 summarises the parameters of the multiple memory multiprocessor entity.

<table>
<thead>
<tr>
<th>Multiple Common Memory Multiprocessor Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Entity</td>
<td>Specifies the entity to be used for the node</td>
</tr>
<tr>
<td>Network Entity</td>
<td>Specifies the entity to be used for the network</td>
</tr>
<tr>
<td>Memory Entity</td>
<td>Specifies the entity to be used for the memory system</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>Controls the number of nodes attached to the network</td>
</tr>
<tr>
<td>Number of Memories</td>
<td>Controls the number of memory entities attached to the network</td>
</tr>
</tbody>
</table>

Table 5.6: Parameters of the multiple common memory multiprocessor entity
The second approach to using multiple memory entities is to distribute them to the nodes of the network. This approach will be discussed in more detail in the next section.

5.2.8 Distributed Shared-Memory Multiprocessor

Distributed shared-memory multiprocessors have a similar structure to that of the message passing machines outlined in Chapter 2. Each node of the network has its own associated memory entity. The basic structure of this type of architecture is shown in Figure 5.7.

![Figure 5.7: A distributed shared-memory multiprocessor](image)

This configuration of multiprocessor allows the system to be scaled to larger numbers of processors without the contention problems of the simple multiprocessor described in Section 5.2.6. It also improves performance for data that is either private or only accessed by one processor, as the data can be located in the local memory entity, enabling the processor to retrieve the data from its local memory without accessing the network or involving any other nodes.

To implement this architecture an extra entity is needed to control access to the network. The network interface entity will be located between the local bus and the...
global network, controlling all packets that the must be sent to and received from the network. The design of this additional entity will be discussed in Section 5.2.9.

Having created an entity that connects the local bus to the global network, a simulation entity to represent the distributed shared-memory system illustrated in Figure 5.7 can be created. This entity is very similar to the one used for the multiple memory component described in Section 5.2.7. The parameters associated with the distributed shared-memory entity allow the designer to change the entity to be used for a node on the global network (this is a different node from the one connected to the bus in Figure 5.7), the network to be used and the number of nodes connected to it. The structure of the node of the global network is much more complicated than the one associated with the multiple memory entity system. It is constructed from a lower level node, a memory, a bus and a network interface entity. This entire structure is then passed as a parameter to the distributed shared-memory multiprocessor entity and used to construct the whole system.

Table 5.7 summarises the parameters of the distributed shared-memory multiprocessor entity.

<table>
<thead>
<tr>
<th>Distributed Shared-Memory Multiprocessor Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Entity</td>
<td>Specifies the entity to be used for the node</td>
</tr>
<tr>
<td>Network Entity</td>
<td>Specifies the entity to be used for the network</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>Controls the number of nodes attached to the network</td>
</tr>
</tbody>
</table>

Table 5.7: Parameters of the distributed shared-memory multiprocessor entity
As with the multiprocessor structures discussed previously, this one also could have performance limitations for certain types of application. If an application was sharing data with only a small number of processors and no others, it might be useful for those few processors to share a memory entity instead of having to share their data over the network. The need to allow multiple processors to share a common local memory has resulted in the emergence, commercially, of clustered systems. Figure 5.8 illustrates a clustered distributed shared-memory multiprocessor.

![Clustered Distributed Shared-Memory Multiprocessor Diagram]

**Figure 5.8: A clustered distributed shared-memory multiprocessor**

A simulation model of this configuration of multiprocessor can be constructed using the distributed shared-memory entity already discussed. A complete multiprocessor system (similar to those discussed in Section 5.2.6 but with an added network interface entity) can be passed as the node entity argument of the multiprocessor entity.

A whole variety of different configurations of multiprocessor can now be constructed using these high-level entities, by passing to them different systems as node, network and memory entity arguments. For example, extra levels of hierarchy can be added to the system by repeatedly passing complete multiprocessor systems as the node entity parameter of multiprocessor entities.
5.2.9 Network Interface Entity

The network interface entity was briefly mentioned in the previous section. The function of this entity is to allow the node to communicate over the external network. It receives and sends messages from both the external and internal networks. Figure 5.9 illustrates the position of the network interface entity relative to the node and external network.

![Diagram of network interface entity](image)

**Figure 5.9:** Position of the network interface entity

The other major function of the network interface entity in shared-memory machines is to maintain system coherence. The multiprocessor system contained within the node is responsible for maintaining its own coherence, for example, the internal network could be a bus running a snoopy coherence protocol. However, when data is required that is not located in the internal memory system, the request has to be forwarded to the appropriate node *via* the external network. It is the function of the network interface component to ensure that the data fetched from the remote node is correct and up-to-date. Similarly, any requests involving accesses to a remote node's memory system in which copies of the requested data are held locally in one of the internal node's caches, must also be dealt with correctly. To make possible these
coherence operations over the external network, a separate cache coherence protocol must be running at this level to maintain a coherent system. There may now be multiple coherence protocols executing within the same system; the network interface component must therefore act as an interface between the two, possibly different, protocols that are used over the internal and external networks.

From a design point of view, the network interface entity is not a key architectural component; it is required to enable hierarchical multiprocessors to be constructed. The associated parameters are included only to enable it to fit into different architectural configurations that are created around it.

The component must have parameters that allow the internal and external protocols to be changed easily. This is necessary, as a key feature of the internal multiprocessor nodes is the ability to change the cache coherence protocol used. As the network interface is responsible for interfacing between the internal and external protocols, it must also be aware of the internal and external protocols in use, enabling its actions to be tailored accordingly. This results in the need for two parameters that inform the entity of which protocols are being used internally and externally.

The next parameters involve the amount of information that can be sent from the network interface (either internally or externally) in one cycle. The networks around the entity have the ability to change the amount of data that can be dealt with in one cycle; the network interface must be aware of this amount, enabling the amount, of data sent to and waited for by either network to be adjusted appropriately. This requires two bus width parameters that specify the amount of data that can be transferred in or out.

The final parameter deals with the delay associated with processing data, either sending or receiving. It represents the time that must elapse between receiving data and sending out an appropriate response.
Table 5.8 summarises the parameters of the network interface entity. Section 6.6 details the implementation of the network interface.

<table>
<thead>
<tr>
<th>Network Interface Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Bus Width</td>
<td>The width of the connection to the internal network</td>
</tr>
<tr>
<td>External Bus Width</td>
<td>The width of the connection to the external network</td>
</tr>
<tr>
<td>Interface Delay</td>
<td>The time it takes the network interface to process either a request or data</td>
</tr>
<tr>
<td>Internal Coherence Protocol</td>
<td>The protocol to be used within the node to maintain a coherent system</td>
</tr>
<tr>
<td>External Coherence Protocol</td>
<td>The protocol to be used outside the node to maintain a coherent system</td>
</tr>
</tbody>
</table>

Table 5.8: Parameters of the network interface entity

The network interface entity also requires communication ports to allow requests and data to be received and sent out to the internal and external networks.

5.2.10 Multiprocessor Model Design Summary

The previous sections have discussed the design of the entities that are required to construct a shared-memory multiprocessor system. The various parameters that each entity requires to provide the designer with enough flexibility to explore a large design space are also discussed. Various ways in which these entities can be connected together to form different multiprocessor entities are also presented. The different configurations that are discussed are by no means exhaustive, and the basic entities described can also be used to construct other forms of multiprocessor architecture. NUMA style architectures have been focussed on, however COMA style architectures and message passing machines could also be constructed.
(although extra parameters may need to be added to some components to provide the
designer with the required flexibility). Section 9.3 discusses how the model and
framework could be extended to include these different styles of multiprocessor
architecture.

The next chapter deals with the implementation of these different entities and
architectures. It also discusses the different parameter options that are supported for
the more complex parameters, for example, cache coherence protocols.
Chapter 6

Implementation of the Multiprocessor Model

This chapter presents a detailed discussion of the implementation of the multiprocessor model, which includes a description of the values supported by each of the parameters of all the entities. The extensions to the HASE simulation environment that improve support for this model are discussed in the next chapter.

The multiprocessor architecture is described using EDL and includes a description of the main entities, how they are connected together and the data structures that they use. As described in Section 4.2.1, an EDL description can be split into five parts: the preamble, parameter library, global parameters, entity library and entity layout. The parameter and entity library sections of the EDL description will be the focus of this chapter, along with the behavioural descriptions. The preamble contains general project information and is of no real interest. The global parameters section describes the parameters that can be accessed by all the entities of the system. Global parameters are not used in this model; apart from being poor programming practice, their use restricts entity reuse. This is because new entities may need to be aware of the global parameters, and entities created for this model may not easily be transferred to other models as they may rely on the existence of a particular set of
global parameters. The other section not discussed is the entity layout section as it contains only a single entity in most of the systems described. A complete EDL description of a distributed shared-memory multiprocessor can be found in Appendix B.1.

6.1 Entity Communication

Simulations created in HASE use the HASE++ library to provide the necessary simulation functions, for example send, wait and hold. HASE++ is a discrete-event simulation package, meaning that the simulation entities wait for the occurrence of an event that affects them. At this point they perform some processing (which may be dependent on the type and contents of the event), followed by a possible generation of an event of their own, before waiting for the next event. The events fall into two basic categories, time outs and communication events.

Time outs occur when an entity executes a hold instruction, which has the effect of stalling the entity for a specified period of time. When the specified length of time has elapsed, an event is sent to the holding entity causing it to continue its operation.

The second type of event, a communication event, is caused by one entity sending a message to another entity. If the receiving entity has executed a wait instruction, an event is generated which causes the entity to receive the sent data, allowing it to continue its operation. If no wait instruction has executed, the event is stalled until the receiving entity executes a wait. The event is then activated and the entity receives the message and continues executing.

It is the communication events that are of interest in this section. In HASE, communication between entities is performed through ports and links. The ports attached to a particular entity are a part of the entity's EDL description and are defined as being either source or destination ports. Source ports may send and receive messages but destination ports may only receive messages, allowing uni- and
bi-direction links to be defined. Port definitions also contain a message type that defines the type of messages that may be communicated through the port. A link specifies a connection between any two ports. Figure 6.1 shows a section of EDL illustrating the different types of port and how they are linked together and Figure 6.2 illustrates these different types of ports.

ENTITYLIB {
  ENTITY ExComp {
    DESCRIPTION ("An example communicating component")
    PARAMS ( )
    PORTS ( PORT ( IN , MessageType , PortIcon );
            PORT ( OUT , MessageType , PortIcon );
            PORT ( InOut , MessageType , PortIcon );
    )
  }
}

LAYOUT {
  LENTITY ExComp Comp1 (DESCRIPTION("Component 1"));
  LENTITY ExComp Comp2 (DESCRIPTION("Component 2"));
  CLINK(ExComp.Compl[OUT]->ExComp.Comp2[IN],1);
  CLINK(ExComp.Comp2[OUT]->ExComp.Compl[IN],1);
  CLINK(ExComp.Compl[InOut]->ExComp.Comp2[InOut],1);
}

Figure 6.1: Example EDL description of ports and links

Figure 6.2: Representation of the EDL description in Figure 6.1

The message types associated with a port define the structures that can be sent through that port. These structures are defined in the EDL by the designer and must cover all the different messages and methods of communication used by the architecture. EDL provides a mechanism for grouping the different message types together to enable a port to send or receive different types of messages.

The first type of message that is required in the multiprocessor simulation is a request; this can be divided into three basic types: read, write and copy-back. Read and write requests originate at the processor and are passed through the system to the
appropriate cache or memory entities. Copy-back requests originate at the cache and are used to write modified data back to a memory entity when it is removed from a cache.

All three types of request have similar requirements for the contents of the message. Reads require the address of the data to be read, whereas writes and copy-backs require the memory address of the location to be updated, as well as the value of the data written. The amount of data that can be moved around is not fixed and is dependent on the width of the connection between the two entities that are communicating; this is equal to the appropriate bus width parameter of the sending entity. The difference between write requests and copy-back requests is that writes only update a single location, whereas copy-backs update a complete block, starting at the address specified. These are not the only fields that are required for a request message. As the message proceeds through different entities in the system and moves between different levels of the multiprocessor hierarchy, different pieces of information need to be recorded. For example, when a request leaves a cache and enters a network or bus entity, the identity of the sending cache needs to be recorded in the message to enable acknowledgments or data to be returned to the correct cache. Request messages must also carry a destination identifier when networks other than buses are used for interconnection. The next part of the message structure allows a unique message identifier to be assigned to each request enabling the system to match up requests with acknowledgments and data. The final part of the request message structure specifies the allocation policy of the memory hierarchy entity that the request last passed through and is specific to write and copy-back requests. This enables lower levels of the memory hierarchy to determine if the higher level is expecting data to be returned after an update has been performed. For example, a cache that was using a write allocate policy would set the allocate part of the message structure for any write request that was sent to memory, informing the memory entity that it required data to be returned. Due to the similarity in message structure for all three of these event types, the same structure is used for all of them. This structure is illustrated in Figure 6.3, and the EDL definition can be found in Appendix B.1.
Now that a request message type has been defined, a further message type needs to be created that is capable of responding to requests. There are two types of response to a request. The first, result message type, sends back some data in response to a read request or a write or copy-back request that states that the higher level entity requires data to be sent once an update is complete. The second, acknowledgement message type, is used when no data is required to be sent back but is required in order to indicate that an update performed by a write or copy-back request has been completed.

The result message type is used to pass data back up the memory hierarchy, therefore the message structure must contain a field for data. As with request messages, the amount of data passed is determined by the width of the connection between the two communicating entities. The result messages also need a field that indicates which cache was the sender of the request to enable the data to be returned to the correct entity. The unique message identifier that was assigned to the request is also required as it enables the data to be matched up with the correct request. The final field specifies the source of the data; it is used later on when cache coherence protocols are in operation and indicates whether a memory or a cache entity supplied the data. In contrast to this, the acknowledgement message type only requires fields to indicate which cache sent the request and one for the unique message identifier. Figure 6.4 illustrates these two message structures; their EDL definitions can be found in Appendix B.1.

These three message structures cover all the basic communication requirements of the multiprocessor model. The other message structures used for specific types of
communication, for example cache coherence protocol messages, will be described later.

![Diagram of message structures](image)

(a) Result message structure

(b) Acknowledgement message structure

**Figure 6.4:** The message structures used for results and acknowledgments

The final area to be described is how message structures are assigned to ports and how multiple message types can be assigned to a port. This is achieved through the use of HASE link parameters. These allow structures to be combined and each structure to be given a unique tag; this complete structure is then assigned to a port in its definition. Figure 6.5 contains a sample section of EDL to illustrate this feature.

```edl
LINK ( t_MemoryLink , [ (REQ_READ , RSTRUCT (t_Request, RReq) ), 
(REQ_WRITE , RSTRUCT (t_Request, WReq) ), 
(COPY_BACK , RSTRUCT (t_Request, CB_Req) ), 
(RESULT , RSTRUCT (t_Result, Res ) ) ] );
PORT ( IN , t_MemoryLink , PortIcon); 
```

**Figure 6.5:** Sample EDL for link parameter definition and use

The link definition, `t_MemoryLink`, groups two different message structures, a result and a request. There are three request structures added to the link, each with their own unique tag to identify the type of request that has been sent. The port definition then includes a reference to this link type in its definition, which defines the types of messages that can be sent through the port. All of the link definitions appear in the parameter library section of the EDL.
This completes the discussion of entity communication in HASE and the structure of the message types used by the multiprocessor. The remainder of this chapter details the implementation of each of the entities and their associated parameters. The implementation of the individual entities is described first, followed by a description of how these entities are combined to form the systems outlined in Chapter 5.

### 6.2 Processor Entity

As discussed in Section 5.2.1 the processor entity has been chosen as execution-driven, and must also provide support for any synchronisation primitives that are required. Both of these require that the application be selected prior to implementation of the processor entity, as the application and its required synchronisation primitives actually form the bulk of the implementation.

The aim of this project is to develop a multiprocessor simulation model that, when combined with a simulation framework, will provide a system that enables designers to explore large portions of a massive design space with a minimum of effort. In order to test and evaluate the multiprocessor model, a standard benchmark has been used as the application program.

There are several parallel benchmark suites that have been developed to help designers test and experiment with multiprocessor systems, including several that are specifically aimed at shared-memory machines. The SPLASH [Sin91] and SPLASH-2 [Woo95] program sets developed at Stanford contain a set of programs from the scientific, engineering and graphics domains. The Numerical Aerodynamic Simulation (NAS) Parallel Benchmarks [Bai95] were developed to evaluate the performance of computing systems for workloads typical in NASA. The programs contained in these sets are smaller than complete applications and concentrate on some smaller problems that must be performed efficiently if any reasonable performance is to be gained from running the complete application. Comparing results gained from running these programs produces valid results, as long as care is
taken not to assume that good performance achieved for a benchmark will automatically mean good performance will be obtained for large applications.

One of the programs, *lu*, from the SPLASH-2 set was chosen as the program to be used to drive the simulations. Results from the SPLASH-2 programs are frequently used in the literature when comparing different approaches, solutions or machines. Using an application from this set is therefore reasonable, as the behaviour of the programs are well understood, making the interpretation of results easier; these results can also be compared to results obtained from other simulations. Extreme caution must be exercised when making comparisons with other results that have been published; unless all things are equal, widely different results could be observed, as slight differences in parameters could have significant effects on the performance of the system.

The *lu* program was selected for several reasons. Firstly, the source code is relatively short and requires less conversion to fit into the simulation. Secondly, the input to the program can be scaled to very small or very large problem sizes with ease, enabling the program to be tailored to the multiprocessor being studied and the time available to perform the experiments. Finally, a definite result is produced that can be checked at the end to see if the simulated machine behaved correctly.

The *lu* program factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense $n \times n$ matrix is divided into an $N \times N$ array of $B \times B$ blocks (where $n = NB$) to exploit temporal locality on a sub-matrix level. The elements of the matrix are double precision floating point numbers, which each require two 32-bit memory words. The program is written in C++ and uses threads to enable different processors to work on different parts of the problem; barriers and locks are also used for synchronisation.

As the processor entity is execution-driven, the application chosen is coded into the behaviour of the entity. The program is written in C++ so most of the code can be reused when defining the processors' behaviour. The areas to which particular
attention must be paid are reading from and writing to the shared array, the synchronisation of the application with the simulation, and the implementation of the required synchronisation primitives.

### 6.2.1 Data Placement Policy

The program initialisation starts by allocating portions of shared memory to hold the necessary data structures. These are two arrays, one of size $n \times n$ and one of size $n$; a single value is also needed to hold the next processor identifier. The $n \times n$ array holds the matrix to be operated on and the single dimension array is used to perform the correctness test. Both of the arrays hold double precision floating point numbers so the amount of space required in memory is $2 \times n \times n$ and $2 \times n$.

These shared data structures need to be placed in memory. There are several different approaches to data placement that are available, which fall into two categories, static and dynamic. Static placement policies are determined at compile-time, for example, assigning each element of the data structures to a particular location in a particular memory entity. Dynamic placement policies determine the memory location at run-time, for example, assigning the elements to the next free location in the memory entity nearest to the processor that made the request. Dynamic placement is more realistic and it would also work for any application that is executed on the model; however, it is more complicated to implement. In contrast, static placement is much simpler to implement, although each application would require a different static placement strategy. The model implemented here is driven by a single application, $lu$, (as described earlier) and the flexibility of dynamic placement is not required, therefore static placement has been chosen for its simplicity.

As static data placement is being used, the assignment of data structures to memory locations has to be decided upon. The first case is when only a single memory entity exists in the system. Here the data placement is straightforward, as all the data must reside in that entity. The first array will fill the first $2 \times n \times n$ locations, followed by
the array of size $2 \times n$, with the single processor identifier being placed in the last location of available memory. This layout is illustrated in Figure 6.6.

![Figure 6.6: Placement of LU data structures in a single memory entity](image)

When multiple memory entities are being used the data must be split up, resulting in processors having to access different memory entities to retrieve required data. There are several options available for dividing up the data. The first is to view the different memory entities as a single memory and use the data placement policy used for systems with a single memory entity. This would result in some memory locations with only the first array, some with only the second array and some with none at all, giving an imbalance in the accesses to the different memory entities, which could reduce performance as multiple processors try to access the same memory entity. To reduce contention, the arrays could be split up so that each memory entity has an equal portion of both arrays. There are now two questions that remain to complete data placement. The first is what method to use to divide up the two arrays – rows, columns or blocks (only valid for the first array where there are $N \times N$ blocks of size $B \times B$). The second is whether consecutive portions of the array are assigned to the same or different memory entities (see Figure 6.7 for an illustration of this using rows to divide up the array).
The \textit{lu} program does not assign the blocks to processors in any particular order, so using blocks to drive data placement would probably not improve performance significantly and would only complicate the placement policy. That leaves rows and columns as possible methods for division of the arrays. The loops of the application are written in row order, allowing cache blocks to prefetch the data that could be used in consecutive iterations of a loop; this indicates that rows should be used to divide up the array in preference to columns. The final decision to be made is whether to assign consecutive rows to the same or different memory entities. As the blocks are not necessarily used by the processors local to the memory entity to which they are assigned, neither approach would offer a significant advantage over the other. However, to take advantage of any blocks that are local to the processor, consecutive rows will be assigned to the same memory entity to try to ensure that complete blocks are located in the same memory entity. This results in the allocation of arrays in the memory entity as illustrated in Figure 6.7a. If a different application were being used, in which the placement of data were not so clear, parameters could be introduced into the processor entity to enable the designer to experiment with different data placement policies. The data placement policy in the processor entity is performed by a function that takes an array index and a specification of which array is being indexed and translates this to a memory address.

Another problem with the data used by the \textit{lu} program is that it operates on double precision floating point numbers, which occupy two words in memory. However, the cache is designed around a 32-bit word and the processor entity must therefore
issue two requests to fetch the required data. To make the simulation operate correctly, the memory hierarchy is unable to pass around meaningful numbers (i.e. the numbers contained in the matrix), instead the values passed around are the bit representations of the upper and lower halves of the double precision floating point numbers of the matrix. The processor entity has to concatenate two of these together and treat the result as a double precision floating point number to obtain the required value. This is not difficult to do, but care must be taken to avoid any strange casting errors. These could occur when assigning one type of variable to another as C++ automatically creates the correct representation of the value in the new type. This would prove disastrous here as the two integers retrieved from memory hold the correct representation of the floating point number and any casting would create the wrong result.

6.2.2 Synchronising the Application with the Simulation

The second problem area to be solved was the synchronisation of the application with the simulation. This problem was outlined in Section 3.3.2 and is caused by instructions that do not access the memory system being executed on the host processor. Therefore in terms of simulation time they execute immediately, as they cause no delay to be incurred by the simulated processor, resulting in no delay between successive memory requests. Delays must therefore be introduced into the application code to reflect the delays of the instructions executed between two memory requests. To calculate the delay the actual instructions between two requests must be known, allowing an estimate for the time it would take to execute them to be made.

To determine the instructions executed by the \textit{lu} program, the code was rewritten in assembly language so that the actual instructions executed could be seen. Each of the functions was coded separately to allow the instruction sequences to be found. As the functions are relatively small and only called in a small number of places (usually one place) the instructions used to call a function were not considered as the code could be easily rewritten so as not to include function calls. Although this
changed the execution of the application slightly, the simplified assembly language code was much easier to deal with. Once the assembly code has been created, counters can be inserted into the C++ code to count the different instructions executed between memory requests. When a memory request is to be issued, the counters can be consulted to determine the length of time the processor should be stalled before issuing the request, so allowing the application to be synchronised with the simulation.

Parameters are associated with each of the different instructions executed by the application enabling the length of time that they are stalled to be varied. Figure 6.8 shows the three steps involved in converting a simple function. Figure 6.8a shows the original code for the daxpy function and Figure 6.8b shows the assembly code that represents this function. Finally Figure 6.8c shows the converted code that includes the appropriate delay counters and the calls to the load and store functions, which retrieve the data from the simulated shared memory. The load and store functions include simulation calls that cause the processor to be stalled for the number of cycles indicted by hold_counter. This process is carried out for each of the functions in lu to create the behaviour code for the processor entity. Appendix C contains the assembly code for each of the functions from the lu program.

6.2.3 Implementation of the Synchronisation Primitives

The final problem to be solved in implementing the processor entity is the provision of the synchronisation primitives required by the application. The synchronisation primitives provided by the system are specified by a parameter of the processor entity, so the implementation must be performed in such a way that different implementations can easily be selected.

The object oriented nature of C++ provides a suitable method to accomplish this. The synchronisation primitive is supplied as a C++ object, which includes appropriate functions, for example, a barrier primitive object would include a function to indicate that a processor has arrived and a spin-lock primitive object
would include functions for acquiring and releasing the lock. Different implementations of a barrier, which are contained in different files, would implement the *arrived* function in different ways. The value of the parameter of the processor entity is then used to indicate the file containing the implementation to be used; this file can then be linked into the simulation at compile-time. The details of this process are discussed in more detail in Section 7.2.

```
for (i=0; i<n; i++) {
    a[i] += alpha * b[i];
}
```

(a) Original daxpy code used in *lu*

```
LI i 0
StartI:  SUB tmp n i
         BLEZ tmp EndI
         ADD b_addr i base_b
         LD b b_addr
         ADD a_addr i base_a
         LD a a_addr
         MULTD tmp alpha b
         ADDD a a tmp
         SD a a_addr
         ADD i i 1
         BR StartI
EndI:     FUNCTION COMPLETE
```

(b) Assembly language code for daxpy

```
hold_counter+=LDI_delay;
for (I=0; i<n; i++) {
    hold_counter+=SUB_delay+CBR_delay;
    hold_counter+=2*ADD_delay;
    b_i=Read(b[i]);
    a_i=Read(a[i]);
    hold_counter+=MULTD_delay_ADDD_delay;
    Write(a[i], alpha * b_i);
    hold_counter+=ADD_delay+UBR_delay;
}
```

(c) Behavioural code included in execution-driven processor entity

**Figure 6.8:** Steps involved in the conversion of the daxpy function

The *lu* program uses two synchronisation primitives to ensure correct operation, a spin-lock and a barrier. The spin-lock is used to control access to the variable that indicates the identifier of the next free processor, allowing only a single processor to
read and write to this variable at any time. The barrier primitive is used to synchronise between the different stages of the algorithm, ensuring that the previous phase has been completed by all the processors before they move onto the next phase. Both of these primitives need space allocated in shared memory as well as atomic operations that can examine and modify memory locations without being interrupted by another processor.

Three functions are needed to implement a spin-lock, a constructor that creates the primitive and initialises any variables used, an acquire function that is called by a processor entity when it wants to acquire the lock, and a release function that indicates that a processor entity no longer needs the lock. There are many different algorithms for implementing a spin-lock, including test_and_set lock, ticket lock and queuing lock (Meller-Crummey and Scott [Mel91] survey different lock algorithms). Of these, the ticket lock is more efficient than the naïve test_and_set lock, but still relatively simple and was therefore chosen for use here. The basic functions for this type of spin-lock are shown in Figure 6.9 (based on Meller-Crummey and Scott [Mel91]).

```cpp
void ticket_lock::Acquire() {
    my_ticket = fetch_and_increment (next_ticket);
    while (1) {
        sim_hold(1.0);
        if (now_serving == my_ticket) {
            return;
        }
    }
}

void ticket_lock::Release() {
    now_serving = now_serving + 1;
}
```

Figure 6.9: Ticket lock
These basic function definitions can be translated directly into code that can be used in the simulation. However this implementation does not use the simulated shared-memory to store the synchronisation variables (\texttt{next\_ticket} and \texttt{my\_ticket}), they are actually stored as variables on the host processor. To enable the synchronisation calls to access the simulated shared-memory, a second implementation is needed. This calls the read and write functions used by the processor to access the synchronisation variables, causing them to be fetched from the simulated shared-memory and not from the memory of the host machine. To do this, the address of these variables in memory must be known and, as static data placement is being used, this decision has to be made before running the program. There is plenty of space in the memory entities (see Figure 6.6) so placing the variables here is no problem.

Both of these implementations of the ticket lock code are correct. The second implementation produces more accurate performance results for the system being studied but takes longer to simulate as the processor entities continually read from the simulated memory until the lock becomes free. In the first implementation the simulated memory is not accessed at all when acquire and release operations are performed, so speeding up the simulation time. To allow the designer to experiment with these two possible implementations, they can be placed in different files and appropriate values of the processor entity parameter created, allowing the implementation to be changed simply by changing the value of the parameter.

The barrier is implemented in a similar manner, but it only requires two functions, a constructor to create the barrier, and initialise any variables used, and a function to indicate that a processor has arrived at the barrier. As with spin-locks, there are many possible barrier algorithms that could be used, for example sense-reversing centralised barrier, distributed tournament barrier and distributed dissemination barrier (Meller-Crummey and Scott [Me91]). For this project, it is not necessary to implement the most efficient barrier possible, only one that works and demonstrates that it is easy to change to a different one. The algorithm used is the sense-reversing
centralised barrier and the basic functions are shown in Figure 6.10 (based on Meller-Crummey and Scott [Mel91]).

```cpp
barrier::barrier() {
    shared int count = 0;
    shared bool sense = true;
    int local_sense = true;
}

void barrier::Arrived() {
    local_sense = ! (local_sense);
    if (fetch_and_increment (count)==p) {
        count = 0;
        sense = local_sense;
    } else {
        while (sense!=local_sense) {
            sim_hold(1.0);
        }
    }
}
```

**Figure 6.10**: Sense-reversing centralised barrier

As with spin-locks, these functions can be translated directly into code that can be incorporated into the simulation, but the `count` and `sense` variables would not be located in the simulated shared memory. To include these they need to be statically placed in shared memory so that the load and store functions of the processor entity can locate them. This provides two different implementations of the barrier that can be chosen as alternatives, using a processor entity parameter. A third implementation was also created that removed the busy-waiting and used a form of interrupt instead. However it did not use the simulated shared memory for storing the variables and was used to speed up the simulation further as the processors waiting at the barrier were not executing any instructions.

The processor only has one synchronisation parameter so files need to be created that contain the appropriate combinations of these implementations to cover the experiments that are to be performed. More parameters could be introduced into the processor entity to allow arbitrary combinations of the different synchronisation primitives, but these were not needed here as only two primitives with a small number of implementations were used.
6.2.4 Other Implementation Issues of the Processor Entity

The last significant feature of synchronisation is that both the spin-lock and barrier mechanisms require an atomic `fetch_and_increment` operation that increments a memory location and returns a value, all with a single memory request. To enable this to be performed, a new message structure is required. This new message has almost the same structure as a request message; the only exception is that it requires an extra field that specifies the operation to be performed on the memory location, for example `fetch_and_increment`. This field enables different synchronisation operations to be implemented using this message type, for example, `test_and_set` and `compare_and_swap`, by specifying a different value for this field.

This completes the description of the implementation of the `lu` program; the remainder of this section deals with the remaining parameters of the processor entity that were discussed in Section 5.2.1. The bus width parameter is of minimal interest as it only affects the amount of information that can be sent from or received by the processor entity at one time and requires a small program loop to accomplish this. The only other parameter associated with the processor entity is one indicating the memory consistency model to be used.

Memory consistency models are included in the processor entity implementation in the same way as synchronisation primitives, i.e., through the use of an object with a defined interface but variable implementation. Memory consistency models affect the point at which the processor is allowed to issue a read, write or synchronisation request. To reflect this, the memory consistency model object requires two functions; the first is called before a request is issued and the second is called after the request has been issued, thus enabling the processor to be stalled if necessary (depending on the number and types of any outstanding requests).
The memory consistency model implemented is a cross between sequential consistency and a form of weak ordering (see Section 2.4 for a description of these two models). The consistency model object contains a variable that keeps track of the number of outstanding requests. A parameter has been added to the processor entity that specifies the maximum number of requests that can be outstanding at any one time; this is then compared to the current number of outstanding requests. If the number outstanding is less than the allowed number, the request is allowed to proceed. If the number outstanding is more than the allowed number the request is stalled. The type of the request is passed as a parameter to the function as this may have an impact on the decision (depending on the consistency model being used). For the consistency model implemented, all outstanding synchronisation requests must have been completed before entering the barrier or lock, and the barrier or lock must have completed any variable updates before the processor is allowed to continue. Although this memory consistency model is relatively simple, more complicated ones could be implemented and selected by changing the value of the memory consistency parameter of the processor entity.

To complete the processor entity, two extra parameters were added that are specific to the lu program. The first specifies the size of the matrix and the second specifies the size of the sub-block to be used.

6.3 Cache Entity

The implementation of a basic cache entity that can be used with the single processor system (described in Section 5.2.4) is relatively straightforward. Complications arise when the ability to change cache coherence protocols is introduced.

The cache entity is built around an array of cache lines (see Section 5.2.2 for description of contents of the cache line), the number of which is controlled by the cache lines parameter. The cache line is also of variable size as the amount of data in a cache line is specified by the block size parameter, which indicates the number
of 32-bits words in each line. The size of the cache, in bytes, is therefore determined by multiplying the number of lines by the block size and multiplying the result of this by 4 (4 bytes per word).

The parameters that allow well known cache design trade-offs to be examined (allocation policy, write policy and replacement policy) all support standard alternatives, for example, the allocation policy can be either write allocate or no write allocate, the write policy can be either write-through or copy-back and the replacement policy can be either random, least recently used or round robin.

The other common cache design trade-off is the associativity of the cache. The cache entity uses an integer to represent the associativity, with 1 representing a direct-mapped cache, 0 representing a fully associative cache and any other integer representing the number of lines per set, i.e., a value of 2 would produce a 2-way set associative cache. A fully associative cache can also be represented by a value that is equal to the number of lines in the cache.

6.3.1 Cache Coherence Protocols

The cache coherence protocol used can have a significant impact on performance, so the ability to change easily the protocol to be used is a crucial aspect of a shared-memory multiprocessor. The implementation discussed in this section provides an interface to the cache that enables different protocols to be inserted into the cache with a minimum of effort.

The method used to specify and include a protocol into the simulation is the same as that used for synchronisation primitives (see Section 6.2.3). The value of a particular parameter indicates which protocol is to be included when the simulation is generated and compiled. This process is discussed in more detail in Section 7.2.

The cache coherence protocol has a significant influence on the behaviour of the cache entity, for example, it can change which operations the cache performs when
requests are received. The protocol may also require the cache to deal with other types of message when communicating with the other levels of the memory hierarchy, for example, the messages may no longer be restricted to read, write and copy-back requests, and other messages may now have to be handled, including invalidate and read-for-ownership.

The problem with enabling the cache coherence protocol to be changed easily is that it has such an integrated role in the operation of the cache. The behaviour that is specific to the protocol therefore has to be separated from the general cache behavioural code; this enables the code to be separated, allowing different implementations to be swapped in easily. To perform this separation, the key points at which the appropriate parts of the protocol are executed need to be identified. The obvious points at which the protocol must intervene are when messages arrive at the cache and when messages are sent from the cache. At these points the protocol needs to monitor these messages to determine if any further action is required. An example of this is when a cache is using a protocol that does not use write requests to deal with writes that miss in the cache. The protocol could use a special form of read instead, which must replace the write before the request is sent to the next level down in the memory hierarchy. The other area in which the protocol must operate is when the cache accesses a cache line, as the protocol will probably use more of the status bits than the standard cache, which only requires one bit to indicate whether the data is clean or modified. The state of these extra bits could result in a protocol action depending on the type of access, so the protocol needs to be called at these times as well. This results in three areas in which the protocol must be called – when messages are received, when messages are to be sent and when the state of the cache is to be updated due to a read or write.

Now that the areas that are affected by the coherence protocol have been identified, an implementation can be constructed that separates the actions of the protocol from the operation of the cache. The first two areas mentioned, sending and receiving, are both performed in the cache by calls to HASE functions. To send messages, HASE provides a group of functions that can be used to send the various types of message;
and several functions that retrieve the next message are also provided. These functions enable the points at which the cache sends and receives messages to be identified easily. However, the third area (updating the cache) does not correspond to a HASE function call and it is therefore slightly more difficult to identify the areas of the cache operation where the coherence actions may need to be performed. Whenever the cache accesses a line, either for a read or write (for both hits and misses), the update function must be called to ensure that the cache line ends up in the correct state and that no further coherence actions need to be performed. This is necessary as each protocol may use the status bits in a different manner. This results in a set of functions that must be implemented by the coherence protocol in order to ensure that the multiprocessor operates correctly, one to send each type of message, one for each of the different methods of receiving a message and at least one function to update the state of a cache line.

The obvious way to implement the send functions is to change the functions provided by HASE to include any necessary protocol actions. Each would be in a separate file containing its own definition of the send routines. These routines could then be compiled and linked into the simulation depending on the value of the parameter of the cache entity. Although simple, this approach has several minor flaws that resulted in an alternative solution being sought. The first is that the send functions provided by HASE are used by all the entities in the system to send messages, therefore code would have to be inserted into the functions to determine if the entity that called the function required coherence actions to be performed, for example, a processor sending a request is not influenced by any protocol operating in the memory hierarchy. These checks would result in a large amount of unnecessary processing that could significantly slow down the execution of the simulation. The second reason is that different entities that are running the same protocol require a different set of actions, so functions would require code to ensure that the correct protocol code was executed, which could again slow down the execution of the simulation. The final, and probably most important reason is that it is very difficult to allow an arbitrary number of protocols to be included using this approach. As there is only one set of send functions, only one protocol file can be included.
Implementation of systems that use multiple protocols would require files that contain multiple protocols in the same function definitions. This is clearly unacceptable as adding a protocol would require a change to every file and the number of files needed to represent all of the combinations, when more than a couple of protocols were required, would grow very rapidly. Without even considering the other two cases where protocol actions are needed, this approach was obviously unsuitable and a different method for implementing the protocols was needed.

The second approach considered was to use separate send functions for each of the entities that perform protocol actions, which could be included in the entity class definition. The calls to the functions provided by HASE within the entity behaviour would be replaced by the calls to a defined set of send functions, for example, send_REQ_READ would be replaced by a call to protocol_send_REQ_READ. These functions would then call the HASE supplied functions if appropriate. This approach overcomes the first two problems of the previous approach, i.e., only the appropriate entities call functions that contain protocol code, as the provided send functions remain unchanged and functions specific to different entities can be written, removing the need for code to determine the type of entity that called the function and then execute the appropriate protocol code. However, the problem of including multiple coherence protocols in the same multiprocessor system simulation still exists, so a third method was explored.

The third and final implementation used C++ objects to implement the protocol functions. A different object was created for each type of entity, overcoming two of the problems of the first solution. The constructor for the object could take as an argument the protocol to be used by the calling entity, enabling different protocol objects to be created at run-time that are dependent on the value passed. This approach overcomes all three problems and was therefore implemented in the multiprocessor simulation model. The cache protocol object therefore includes an implementation of all the required functions (outlined earlier), with the different implementations of the object and its functions in different files, one for each of the possible protocols. These different implementations can be included in the same
simulation without any conflicts because they each implement a different cache protocol object, for example, IllinoisCacheProtocolObject and MESICacheProtocolObject. The cache can use any of these different protocol objects because they are all derived from a basic cache protocol object (i.e. CacheProtocolObject).

Before discussing any of the protocols it should be noted that the protocols require their own message structure for communication. This is because the action that is being performed has to be included in the message. This is different from simple read and write requests where the operation is not included in the message as there are only three basic requests and these are the same for each simulation. Protocol operations differ from protocol to protocol, so the action has to be included in the message. This results in a message structure that is the same as that used for synchronisation messages, which also require the synchronisation operation to be included in the message.

Although each protocol uses the same basic functions, each one uses these functions in a different manner. The protocol descriptions below describe the entire protocol, including any significant memory and bus operations, in order to give a complete overview of each one. However, only the implementation of the cache operations is discussed in detail in this section; discussion of the implementation of memory and bus protocol operations are in Sections 6.4 and 6.5 respectively. There is one assumption that has been made when implementing the coherence protocols. That there is only one level of caching between the processor and bus. This simplifies the protocol implementation, as protocol messages do not have to be passed further up the memory hierarchy. Multiple levels can be experimented with in a single processor system, but including them here only adds to the implementation detail without adding a significant amount to the flexibility of the model.
The Classical Approach

The simplest way to ensure cache coherence is to make all caches write-through (as implemented in the Balance multiprocessor system [Tha88]). With this policy in place, read requests issued to the cache are handled in the usual fashion, and all write requests to the cache are passed to main memory as the cache is write-through. If the requested data exists in the cache it will be updated, but the request will still go through to main memory. All the other caches in the system are continually monitoring the bus (or snooping) and if a write request is placed on the bus to a line that they are currently holding, the data held in the cache is invalidated. Subsequent access to data that was invalidated will result in a miss and the correct data being fetched from memory. The main problem with this is that all write requests go to main memory, and therefore use the bus even if no other caches contain the line being written to.

This is a simple protocol to implement in the multiprocessor model developed, as all the send functions of the protocol object have only two operations to perform, request the bus (if required) and issue the request by calling the provided send functions. The update functions also require no work, as this approach does not do anything extra with the status bits of the cache line. The only extra code that needs to be inserted is in the receive functions, which must now be capable of receiving invalidation requests from the bus and then invalidating the appropriate cache line (if necessary).

The excessive use of the bus can be overcome by making the caches use a copy-back policy instead of a write-through policy. However, this introduces problems of its own, as not all writes are presented to main memory, and copies held by other caches have to be maintained in a consistent state through a more complex protocol. For this to be worthwhile, the performance gain from the decrease in bus traffic must outweigh the increase in complexity.
Write-Once

The Write-Once scheme was proposed by Goodman [Goo83] as a new write strategy for caches that combined the easy to maintain approach of the write-through cache with the reduced amount of bus traffic in copy-back caches. The protocol uses four states per cache line to maintain coherency - Invalid, Valid, Reserved and Dirty. The cache deals with read requests in the standard way, regardless of the state of the cache line, with Dirty lines being written back when they are replaced. Write requests (hits or misses) to either Invalid or Valid cache lines are dealt with using a write-through policy, as another cache could hold a copy of the line being written to. Once the request has been completed and any copies in other caches invalidated, the state of the line is set to Reserved, indicating that this cache holds the only copy of the data. A future write hit to a Reserved or Dirty line can proceed without requests being issued to the bus, with Reserved lines being changed to Dirty lines. Other caches requesting a block stored in another cache in the Dirty status causes the stored line to be changed to Valid and the data to be supplied to the requesting cache and main memory.

As with the Classical protocol, very little work needs to be done to implement the send functions of the cache protocol object within the multiprocessor model as the Write-Once protocol is based on standard requests. The receive functions must again deal with invalidate messages, but must also deal with snoop_read messages. These messages are sent from the bus when a different cache issues a read request, and cause the cache to send out the required data if it is held in a Dirty state. The update functions are also more complicated as the protocol uses extra status bits. The pre-update function must ensure that write requests that hit a Valid line are passed onto the bus to invalidate any other possible copies held in other caches before updating the line. The update function must ensure that the state of the cache is correct after the request has been dealt with, taking into consideration its state before the request and the type of request.
MESI

The MESI protocol is very similar to the Write-Once protocol described above; it uses four states to maintain coherence – Modified, Exclusive, Shared and Invalid. The cache handles reads in the normal way, with the state of the fetched data being set to Shared. Write misses result in the data being fetched into the cache and its state being set to Exclusive. Write hits to a Shared cache line also have to be passed onto the bus to enable other copies of the data to be invalidated. Write hits to other states are handled within the cache. Other caches requesting a block stored in another cache in the Modified state causes the stored line to be changed to Shared and the data supplied to main memory, at which point the original request can be satisfied.

The implementation of the cache protocol object within the multiprocessor model developed follows the same lines as the implementation of the Write-Once protocol, with similar operations being carried out in the functions. The differences between the two protocols are in the memory and bus entities (and this is outlined in Sections 6.4 and 6.5).

The advantage of the Write-Once and MESI schemes is their simplicity. However, in an attempt to gain better performance or lower bus utilisation more complicated protocols have been developed.

MOESI

The MOESI protocol is ownership based and uses five states to maintain coherence – Modified, Owned, Exclusive, Shared and Invalid. It also supports cache-to-cache transfers to enable caches to supply data directly to other caches, removing the need to access main memory every time. The cache handles reads in the normal way, with the state of the fetched data being set to Shared. Write hits to a Shared or Owned cache lines also have to be passed on to the bus to enable other copies of the data to be invalidated. Write hits to Modified cache lines are handled within the cache. Any copies of the requested data stored in other caches in the Modified or Owned state
are sent directly to the requesting cache and the response from main memory is disabled.

There is very little work to be carried out in the implementation of the send functions of the MOESI cache protocol object within the multiprocessor model as the standard requests are still used. The receive functions again have to deal with invalidations and snoop reads, although the extra state, Owned, increases the chance of data being supplied by the cache and not main memory. The pre-update function is not used by this protocol; however the update function has the responsibility of ensuring that write requests that hit a Modified or Owned line are forwarded to the bus, as well as ensuring that the state of the cache line is correct after any request.

By introducing the notion of ownership, the MOESI protocol tries to cut down the number of times main memory is used to respond to a request by allowing a cache that is the line’s owner to supply the data instead. This should speed up responses to requests, as the access time of a cache is less than that of main memory. This could in turn reduce contention for the bus, as a cache will have control of the bus for a shorter length of time.

Other protocols have also been proposed that employ the ownership principal to try and improve performance and three of these, Synapse, Berkeley and Illinois are discussed next.

**Synapse**

This protocol was developed by Synapse for the N+1 fault-tolerant multiprocessor [Fra84]. The protocol uses a form of ownership to maintain coherence and requires that the main memory has a set of usage bits (one for each memory block) as well as three states associated with each cache line – Invalid, Valid and Dirty. The usage bits indicate whether a cache holds a modified copy of the data, thus enabling the memory entity to stop itself from responding to requests that are held in a Dirty state by another cache. As well as these status bits, the protocol also uses two different
types of reads for fetching data into the cache – Public and Private. The Public read is the standard read, but the Private read indicates to memory and other caches that the data will be modified and any local copies must be copied back or invalidated.

The cache deals with read requests in the usual way and uses the Public read to request data not in the cache. A write to an Invalid or Valid state causes the data to be requested using a Private read, resulting in the cache having the only copy of the data. Write hits to a Dirty cache line can be serviced by the cache as it is the only cache with a copy of the data. Read requests from other caches for Dirty data cause the cache to copy the data back to memory and to the requesting cache if it was a Public read request, or to the requesting cache only if it was a Private read request.

The send functions of the cache protocol object for this particular protocol are implemented in the multiprocessor model in the same way as those for the previous protocols described, i.e., they request the bus and call the supplied send functions. However, one of the send functions needs to be different as the function to send a write request now has to send a Private read instead. The request is constructed using a protocol message, with the contents of the write request copied into it and the action field of the protocol message being set to Private_Read. The receive functions must be implemented to deal with two types of snoop request, Public and Private, and the update functions must ensure that the cache line is in the correct state after the request has completed. The pre-update function must issue a Private read request if a write hits a cache line that is not modified, to ensure that the copy in the cache is the only copy held in any cache.

Berkeley

The Berkeley cache coherence protocol [Kat85] also uses the idea of ownership in an attempt to reduce bus utilisation and improve performance. The protocol uses four states – Invalid, Unowned, Owned Non-Exclusively, and Owned Exclusively, as well as several extra message types, Read-for-Ownership, Write-for-Invalidation and Write-without-Invalidation. The Read-for-Ownership operation is similar to the
normal read, except that the cache making the request changes to the Owned Exclusively state instead of the Unowned state and copies held by other caches are invalidated. The Write-for-Invalidation operation invalidates any copies held by other caches but does not update main memory. The Write-without-Invalidation operation is used to update main memory when cache blocks are evicted, but any copies held by other caches remain valid.

The cache deals with read requests in the normal manner, with the conventional read operation being used to request data not in the cache. Write misses cause data to be fetched into the cache using the Read-for-Ownership operation, ensuring that it is the only copy, with state set to Owned Exclusively. Write hits to either Unowned or Owned Non-Exclusively cache lines cause the bus to perform a Write-for-Invalidation operation to ensure it is the only cached copy.

The implementation of the Berkeley protocol in the multiprocessor model requires two of the send functions to perform extra work. The function responsible for sending write requests must now issue Read-for-Ownership messages instead and the function responsible for sending copy-back requests must now issue Write-without-Invalidation messages. The receive functions must be capable of accepting the four different types of protocol message used, as well as responding to them correctly. In addition to updating the state of the cache line being accessed, the update function must also issue a Write-for-Invalidation message when a write request hits a cache line in the Unowned or Owned Non-Exclusively state.

**Illinois**

The Illinois cache coherence protocol [Pap84] also relies on a form of ownership and uses four states – Invalid, Shared, Exclusive-Unmodified and Exclusive-Modified. In addition the Illinois protocol requires only two extra message types – Read-for-Ownership and Invalidate. The Read-for-Ownership operation requests data from either main memory or other caches (whichever is the most up to date); followed by
an invalidation of any cached copies. The Invalidation operation invalidates any cached copies of the data.

The cache deals with read requests in the usual way with the standard read operation being used to request data not in the cache. Write misses cause data to be fetched into the cache using the Read-for-Ownership bus operation, ensuring that it is the only copy held in the caches. The state of the data fetched is set to Exclusive-Modified. Write hits to a Shared cache line cause an Invalidation operation to be issued before the cache line is updated and the state is then set to Exclusive-Modified. Write hits to either Exclusive-Unmodified or Exclusive-Modified cache lines are written to the cache only.

The implementation in the multiprocessor model of most of the send functions of the cache protocol object is simple, requiring calls to request control of the bus and to the HASE send functions. The only exception is the write request function, which must issue a Read-for-Ownership message to read the data from the next level of the memory hierarchy and ensure that no other copies of the data exist. The receive functions must again deal with Read-for-Ownership and Invalidate messages. The update function is required to issue an Invalidate message whenever a write request hits a line that is in the Shared state, otherwise write hits can be handled by the cache.

The cache coherence protocols implemented so far are all invalidation based, i.e., they maintain coherence between the caches by invalidating other copies held in other caches whenever these multiple copies may cause stale data to exist. Invalidations can cause an increase in bus traffic as the number of cache misses will increase, resulting in more requests being issued to memory to retrieve data. To overcome this source of performance loss, update protocols have been proposed. Rather than invalidating data held by other caches, lines are updated so that they contain the new value. The final two snoopy protocols that have been implemented are update protocols and are described next.
Firefly

The Firefly protocol [Tha87] uses only three states – Shared, Valid-Exclusive and Dirty, (no Invalid state is used). The protocol does not use any extra bus operations, but does require a special bus line, called the Shared Line, which is used to indicate that other caches are sharing data currently on the bus.

The cache deals with read requests in the normal way, except that on read misses the Shared Line is used to determine the state of the data fetched. Write misses cause the data to be fetched from either another cache or main memory. If fetched from another cache, the new data is written through the cache so that the copies in other caches and main memory can be updated. However, if the data came from main memory the write is only performed in the cache. Write hits to a Shared cache line also causes the new data to be placed on the bus, enabling any other copies to be updated. Write hits to cache lines in the other states can be handled by a write to the cache only.

Very little work needs to be done in the implementation of the send functions of the cache protocol object in the multiprocessor model as the Firefly protocol is based on the standard requests, although write requests are translated into read requests to fetch the data before updating it in the cache. The receive functions now have to deal with Update messages which may cause a cache line to be updated. The pre-update function is responsible for sending a write request to the bus if a write request resulted in a hit on a cache line in the Shared state. The update function is responsible for ensuring that the cache line finishes in the correct state and that an Update message is sent if a write miss caused data to be fetched from another cache.

Dragon

The protocol used in the Dragon computer system [McC85] is similar to the Firefly protocol, but write operations only update other caches, unlike the Firefly protocol which sometimes updates main memory as well. To accomplish this, Dragon uses
four states instead of three. These are Shared-Clean, Shared-Dirty, Valid-Exclusive and Dirty; a Shared Line in the bus is also required by the protocol.

The cache deals with read requests in the normal way, except that on read misses the Shared Line is used to determine the state of the data fetched. No other copies of the data cause the Valid-Exclusive state to be used, otherwise the Shared-Clean state is used. Write misses cause the data to be fetched from either another cache or main memory. If fetched from another cache the new data is written through the cache so that copies in other caches can be updated. However, if the data came from main memory the write is performed only in the cache. Write hits to a Shared-Clean or a Shared-Dirty cache line also causes the new data to be placed on the bus, enabling any other copies to be updated. Write hits to cache lines in the other states can be handled by a write to the cache only.

The implementation in the multiprocessor model of the cache protocol object is similar to that of the Firefly protocol. Write requests are translated to read requests by the appropriate send function, the receive functions have to deal with Update messages as well as the usual Snoop requests and the update functions are responsible for issuing write and Update requests when needed as well as setting the state of the cache line.

This completes the description of the cache protocol objects for all the protocols that are supported by the cache entity. The remainder of the protocol implementations (i.e. the memory and bus protocol objects) are described in the next two sections. There are many other possible protocols that could have been implemented, but the aim of this project is to provide a model that allows exploration of different systems and architectures, not to provide an exhaustive system for evaluating multiprocessors. The protocols that have been implemented cover the basic types of snoopy protocols, ranging from simple protocols, for example the classical approach, through to more complex ownership based protocols like Illinois. A mixture of invalidation and update protocols is also covered. Other cache coherence protocols can be added by creating new cache, bus and memory protocol objects that can then
be included in simulations, allowing more efficient, complicated or experimental protocols to be evaluated.

As can be seen from the EDL definition of the cache entity (in Appendix B.1) there are other parameters, apart from the design parameters outlined in Section 5.2.2, that have been included in the entity. These have been included either to enable the operation of the cache to be monitored or for statistics gathering. The tag, index and block offset parameters show the breakdown of the currently requested address; this can be displayed during a HASE animation to ensure that the cache is decoding the address correctly and accessing the correct location in the cache. The cache state parameter records the access type (hit or miss) and the request type (read or write). The cache action parameter records the action being performed by the cache (read, write or copy-back) and the hits and misses parameters keep a running total of the number of hits and misses, enabling the hit rate to be calculated. The different state parameters are recorded by the simulation in the trace file and can be processed by HASE to give an indication of the time spent in each state.

6.4 Memory Entity

The implementation of the basic memory entity is straightforward. Its functions are to receive requests regarding the data stored in the array (used to hold the contents of the memory) and to return any required data or acknowledge the request’s completion. Most of the memory entity’s parameters, for example the read and write delays, block size, and bus width, are easy to incorporate into the entity’s behaviour. The parameter that is more difficult to implement is the coherence protocol.

The coherence protocol is implemented using the same mechanism used in the cache entity. A memory protocol object is created (one for each protocol that can be included in the simulation) that must conform to the defined interface. A memory protocol object is needed to interpret any messages that are sent to the memory entity that are not standard requests. Some protocols also require the memory to behave in
a different way, for example, the Synapse protocol specifies that the memory contains a set of status bits that indicate which cache entity is the current owner of the cache line. The memory protocol object also allows directory protocols to be implemented as they require the memory entity to perform the coherence actions, unlike the snoopy protocols which rely more on the bus.

The memory protocol object has a similar interface to the cache protocol object. It specifies that a number of send and receive functions (but no update functions) need to be defined for each protocol. Of the protocols described in Section 6.3.1, five of them (Classical, Write-Once, MESI, MOESI and Dragon) do not require the memory protocol object to perform any extra actions, so the function’s only operation is to call the corresponding supplied HASE function. Of the other four protocols, three of them (Berkeley, Illinois and Firefly) only require extra code to be added to the receive functions which converts the requests specific to the protocol into requests that the memory entity can understand. For example, the Firefly protocol occasionally sends Update requests to the memory to update the contents of a particular address; these are translated by the memory protocol object into a write request with the allocation field set to 0 (to indicate that only an acknowledgement should be returned).

The final protocol, Synapse, requires the most complicated memory protocol object of all of these implemented. It requires the receive functions to translate Private read requests into ordinary read requests to enable the memory entity to respond to them. The second responsibility of the memory protocol object for this protocol is to ensure that the status bits (indicating which cache entity is the owner) are set correctly and also queried before requests are forwarded to the memory, to ensure that requests to data owned by a cache are ignored. The send functions also require small pieces of code to ensure that the status bits are used correctly.

The final parameter included in the EDL description of the memory entity (see Appendix B.1) is used to indicate its state. The memory state parameter can be set to read, write, copy-back or idle; it records the state of the memory entity in a trace file.
over the course of the simulation. This can then be processed by HASE to give an indication of how the memory is being used and how busy it is.

6.5 Bus Entity

The basic operation of the bus entity is to forward messages from one entity to another. However, there are complications. Firstly, only one entity is allowed to use the bus at any one time, which means that a mechanism for assigning control of the bus to one of the entities is required. Secondly, the bus is a central entity used to ensure system coherence; this results in many coherence messages arriving and leaving the bus which need to be processed correctly.

The simple bus operations of forwarding messages can be implemented with very little code. Once a message is received, its type and source is examined before deciding which entity it should be sent to. Sending and receiving of messages are performed by the supplied HASE functions. Including the bus width and bus cycle parameters into this implementation is relatively straightforward. The bus width parameters are included in program loops around the send and receive functions to enable appropriately sized messages to be sent and received. The bus cycle parameter is used to stall the bus's operation whenever a message is received, to simulate the time it takes the bus to deal with the message.

To control access to the bus an arbitration scheme is needed; this is used to determine which of the entities waiting to use the bus goes first. The arbitration scheme used by the bus entity is specified using an entity parameter. It uses a mechanism similar to the coherence protocol (see Section 6.3.1) to include different schemes into the simulation based on the parameter value. Only one function needs to be supplied by the arbitration scheme; it examines the outstanding bus requests and returns the identifier of the next entity to which control is to be granted. The bus is implemented around a request and release system, i.e., when an entity wishes to use the bus, it
sends a bus request message, and the bus arbitration scheme then selects one of the outstanding bus requests and sends a bus grant message to the appropriate entity. This entity then has control of the bus, allowing it to send and receive messages. When the entity has finished, it sends a bus release message and the bus assigns control to another entity. The arbitration scheme implemented is round robin, allowing each entity an equal share of the bus’s time. If desired, other arbitration schemes could be implemented, and included in the simulation.

As with the cache and memory entities, the coherence protocol is implemented in the bus by using a bus protocol object. Again this contains the functions to send and receive messages, but no update functions are needed. The bus protocol object is where the bulk of the protocol code for snoopy coherence protocols resides. It has to ensure that when a message is received the appropriate coherence messages are sent to all the relevant entities. This is not what happens in actual hardware; the caches themselves monitor the bus and take actions according to what they see. However, this is not the easiest mechanism to include in a discrete-event simulation, where messages sent to the bus are not observed by any other entities. To model the snooping process, the bus sends out messages to all appropriate entities whenever a message is received. This allows the caches and any other entities connected to the bus to take any necessary actions in response to the message received, allowing coherence to be maintained. The downside to this approach is that the bus protocol object becomes the most complicated part of the coherence protocol implementation.

The problems with implementing the bus do not arise from the sending of appropriate messages to the correct entities, rather, they occur because of the need to ensure that the cache receives the correct data and, if required, an acknowledgement. For example, a cache issues a read request to the bus, which results in appropriate protocol messages being sent to the other caches in the system, as well as to memory, to retrieve the most up-to-date copy of the data. Data could be returned from any or all of the caches as well as from memory; one of these must then be sent to the cache that made the request. This cache could in turn have to copy-back some data that was replaced in the cache, which is then passed on to the memory. Coordinating all
of these messages is the main function of the bus protocol object, as well as ensuring that an acknowledgement is sent to the cache upon completion of a request.

The bus uses two methods for dealing with all the different messages and ensuring that the protocol operates correctly. The first is the message identifier field of the messages. Each new request that the bus receives is assigned a unique identifier and all messages that are generated with a connection to this request (by any entity) are also assigned this identifier. This enables the bus to determine which results and acknowledgements are for which request. By placing the message identifiers in a list of outstanding requests, the bus is also capable of determining if a result is too late and can therefore be ignored, i.e., a response has already been sent for the particular message identifier.

The second method used is an array of counters (one for each cache), which are used to keep track of how many protocol messages have still to be processed before the request can be considered complete. For example, when a read request is sent to the bus, the bus may generate a snoop request for all the other caches, causing the outstanding request array entry for the requesting cache to be set to $n-1$ (for a system with $n$ caches). When a response is received from a cache, the appropriate cache counter is decremented by one. If data is received from memory before the outstanding counter has reached zero, the data is stalled until all the outstanding protocol messages have been dealt with. This ensures that more up-to-date data stored in a cache is returned to the requesting cache and not the older data stored in memory. This array can also be used to generate acknowledgements that indicate completion of a request. For example, in an invalidation based protocol, the protocol may require an invalidation message be sent to all but one of the caches; the acknowledgement is sent to the cache that requested the invalidations when the counter in the array reaches zero (all caches have performed the invalidation).

Most of this work is performed by the receive functions of the bus protocol object. It must intercept all messages that are sent to the bus and generate the appropriate sequence of protocol messages to ensure that coherence is maintained. The amount
of code required to perform this is dependent on the complexity of the coherence protocol. The Classical protocol requires very little code, as all writes proceed through the system to the memory (the caches are write-through). This means that the bus is responsible for invalidating all the copies held in caches whenever a write request is received and for collecting the invalidation acknowledgements. In contrast the Berkeley protocol requires the bus to perform actions on read, Read-for-Ownership, Write-for-Invalidate and Write-without-Invalidate, all of which have a corresponding set of replies. The protocol is further complicated by cache-to-cache transfers, which means that memory has to be ignored when data has been supplied by another cache.

One final parameter has been included in the bus entity for statistics gathering. The bus state parameter records the state of the bus (busy or idle) in the trace file over the course of the simulation. This file can then be processed by HASE to give an indication of how heavily the bus is being used. This measure can be extremely useful when designing a small-scale multiprocessor based around a bus, as it can indicate when there are too many processors attempting to use the bus (i.e. it is saturated) and when adding any more would not improve system performance (and could possibly degrade performance).

6.5.1 Extending the Bus Entity for Distributed Shared-Memory Multiprocessors

The bus described in the previous section works well for systems in which all processors are connected to the same bus as the memory. However, the implementation needs to be extended for it to work with the distributed shared-memory systems discussed in Sections 5.2.7 and 5.2.8. Most of the extensions revolve around the need to release the bus while waiting for responses from remote nodes; they include adding a network interface entity to the arbitration scheme and dealing with protocol messages that must be sent to and received from the network interface. If the bus was not released, the multiprocessor would be extremely slow, as a node would have to wait for the messages to be processed by remote nodes and
responses returned before continuing the request locally and then releasing the bus. Whilst the bus was waiting, all other nodes would be unable to use it, even though it was idle.

The first extension involves adding the network interface to the arbitration scheme, which is a relatively simple task. More complicated schemes could be developed that give the network interface a greater or lesser priority when requesting the bus than the other nodes. These have not been developed as they would not prove anything more about the flexibility of the model; they would only provide extra parameter values with which to experiment.

Extending the protocol handling code to deal with the network interface (and the coherence operations that must be performed externally) required rather more work. Most of this is discussed in Section 6.6, which deals with the implementation of the network interface entity, although the general extensions are briefly outlined here.

All requests that originate with a node connected to the bus must cause a coherence message to be sent to the network interface to allow the external coherence protocol to maintain coherence in the rest of the system. Similarly the bus must be capable of dealing with protocol messages that are sent by the network interface.

The final extension to the bus involves the creation of retry packets, which cause a cache to re-send its request. These are needed as the system may now contain several networks with a small number of caches connected to some of them; this opens up the possibility of requests and protocol messages that refer to the same memory address, but originate in different places, overlapping. This overlapping, if involving modified data, will almost certainly result in an incoherent state. The simulation entities must therefore detect that such a problem has occurred and force all but one of the conflicting messages to be retried. Most of this detection is done by other entities (mainly the network interface); however the bus must be capable of dealing with retry packets and passing them onto the appropriate cache or network interface.
6.6 Network Interface

As outlined in Section 5.2.9 the network interface entity is responsible for passing messages between two levels of the architecture. These two levels may be executing two different coherence protocols, which requires the network interface to translate the protocol messages that move from level to the other. The other main function of the network interface is to detect requests that overlap and cause coherence problems. This was outlined in Section 6.5.1, but will be discussed in more detail in this section.

In the discussion of the design of the network interface entity in Section 5.2.9, it was represented as a single entity with five parameters. However, when the entity was being implemented it became clear that to implement it as a single entity would be very complicated and extremely difficult. The cause of this is the level of parallelism contained in the entity. It must be capable of receiving messages and forwarding messages to the upper and lower networks. This does not appear to cause any problems, but messages received at the upper level and sent to the lower level should not interfere with messages received at the lower level and sent to the upper level, i.e., delays associated with one message should not affect an unrelated message. To achieve correct timing for the different combinations of message routes, without stalls in one message affecting other messages was particularly difficult. If these timing problems are combined with the protocol translation and the detection of overlapping requests, the entity implementation becomes extremely complicated.

The solution to these problems is to split the network interface into four entities, a receiver and a sender for both networks. These four entities, along with their communication links are illustrated in Figure 6.11.
This division of the network interface requires that the parameters be assigned to appropriate entities, for example, the lower level receiver entity does not need to know the width of the upper network.

The first problem to be addressed in the implementation of the four entities is the translation of protocol messages between the upper level and lower level protocols. The first approach considered was to create a set of functions that translated messages from any protocol to any other protocol currently understood by the system. This would require at least one function for each combination of protocols, and every protocol added would require more and more work to keep the list of functions complete. This approach was not taken due to the work involved in creating all the required functions and the problems in extending the code to include more protocols.
The approach that was taken was to use a generic protocol language for an intermediate message structure. Cache coherence protocols in general perform similar sorts of operations, but differ in the terminology used, for example, the Synapse protocol has a Private read whereas the Berkeley protocol use a Read-for-Ownership message. This enables a set of generic operations to be created that can be used to communicate between the two protocols. This would result in two functions per protocol, one to convert the protocol messages into the generic format and one to convert the generic format into protocol messages. The protocol object method (used in the cache, bus and memory entities) is used to allow different protocols to be included in the network interface entities. The network interface protocol object contains two main functions, Send and Receive, that are called by the network interface entities whenever messages are to be sent or received. The Send function takes a generic protocol message as an argument and translates it to the appropriate protocol message before sending it to the network, whereas the Receive function takes a protocol message as an argument and translates it into a generic protocol message before sending it to one of the other network interface entities. The generic message structure, as well as the Send and Receive functions must also be capable of handling ordinary messages for representing requests, results and acknowledgments.

The main implementation issue with the network interface entities was the upper level protocols. The upper network does not necessarily have to be a bus so a snoopy protocol is no longer appropriate; this requires new protocols that rely on directories instead of snooping to be implemented. The directory protocols developed are based on the protocol presented by Censier and Feautrier [Cen78]; this needed to be extended to enable it to deal with (clustered) distributed shared-memory multiprocessors.

There are two basic types of protocol that have been implemented at the lower level, invalidate and update. Each of these requires an upper level protocol to maintain coherence in a similar fashion at the upper level. This is because a cache issuing an invalidation request expects that all other copies in the system will be invalidated;
however an update protocol operating at the upper level will leave copies in remote caches. Two directory-based coherence protocols have therefore been developed to enable systems to be constructed that contain multiprocessor nodes using any of the snoopy protocols implemented.

6.6.1 Upper Level Coherence Protocols

There is much published work on cache coherence protocols, both snoopy-based and directory-based, as well as hybrid schemes. Most of these protocols have been designed and implemented on a flat multiprocessor system; i.e., one in which all of the nodes are connected to the same network, which also connects all of the memory modules. However, some shared-memory multiprocessor systems have been developed that contain hierarchical levels and multiple networks, for example, the DASH [Len92], the SGI Origin [Lau97] and hierarchical bus system created by Anderson and Baer [And93], which also contain coherence protocols designed to work with these multiple networks. However, adapting these specific protocols to work with a general architecture and a general lower level protocol is not possible as they assume particular features, for example, the directory protocol used in the DASH to support coherence over the network assumes that the network interface contains a specialised cache. Two protocols have therefore had to be developed from first principles, using the Censier and Feautrier protocol [Cen78] for guidance on the operations that directory protocols perform.

The biggest problem when designing these protocols is that they do not have direct access to the memory or cache entities, so they have to operate on messages that are sent to the network interface entities. As outlined earlier, this requires that snoopy protocols ensure that messages are also sent to the network interface to inform it of what is occurring at the lower level.

During the description of the upper level protocols the term node will be used to represent an entire lower level system that is connected to the upper network through
a network interface. A node will therefore contain a memory, a bus and at least one processor and cache (illustrated in Figure 6.12).

![Diagram of a node](image)

**Figure 6.12:** The structure of a node

The upper level protocols maintain a directory that contains a record of all nodes that hold a copy of a memory block that they are monitoring. Each directory contains all the information regarding the memory blocks that reside in the node to which it is associated. Any request made by remote nodes for data held in a memory must therefore go through the network interface and its directory to reach the memory module. This enables the protocol to take appropriate actions depending on the state of the directory. To enable the directory to be kept up-to-date and coherence to be maintained, requests from local processors for data held locally also have to be sent to the network interface. A more detailed discussion of the operation of the two protocols is presented in the next two sections.

**Full-Map Invalidation Coherence Protocol**

The first protocol developed was a full-map invalidation protocol (see Section 2.2.4 for a description of directory protocols). The directory contains one line for each
memory block in the local memory entity. Each line contains a bit for every node in the system, which is used to indicate which nodes currently hold a copy of that memory block; a final bit is used to indicate whether the data is held in a modified state. There can be only one copy held by any cache if the data is in a modified state. It is worth noting that if one of the entries in a directory indicates that a node holds a copy of the memory block and the node contains multiple caches, the directory only indicates that at least one of the caches has a copy, not precisely how many.

The protocol operates in the expected manner with no optimisations to try to improve performance. All requests are initially directed at the node that is the owner of the memory block, referred to as the home node. Read requests to data not held in a modified state cause the data to be fetched from memory and not from a cache. The only exception to this is when the request originated at the home node, in which case the low level protocol deals with the response. However, read requests to data held in a modified state causes the request to be forwarded to the appropriate node, which then supplies the data to the requesting cache, as well as writing the data back to memory. Write requests cause all other copies to be invalidated before the data is updated, ensuring that the requesting cache is the only cache to hold a copy of the data.

Care must be taken to ensure that the protocol operations are directed at the correct nodes, that multiple responses are grouped together and that a single response is passed down to the lower level protocol. If this is not performed correctly, multiple acknowledgements of copies of data will be passed to the lower level protocol, which will almost certainly cause the protocol to break down, and operate in an unpredictable manner.

There are potential problems with this approach, as it is possible for requests to be directed to the wrong node for data causing the wrong data to be fetched. This is caused by the delay between carrying out the protocol operation and updating the directory entries. There are two possibilities for updating the directory entries; the directory can be updated and then issue the protocol operations, or the directory can
wait for the protocol operations to complete before updating. In either case, for a period of time, the state of the system is not correctly represented by the directory, which causes a variety of errors. Multiple requests for the same data by different caches in close proximity could cause the wrong data to be supplied to one or more of the caches. Requests could also be directed at nodes that do not contain the required data, for example, a copy-back could have been issued to write the data back to memory, however it may not have arrived at the directory before a request for the data is received, causing the request to be directed at a cache that no longer has a copy. These problems are overcome by detecting requests to the same memory block as requests that are outstanding and forcing them to retry. The mechanism to perform this will be discussed below.

Full-Map Update Coherence Protocol

The second protocol developed was a full-map protocol that supported updates. This is very similar to the invalidation-based full-map protocol, but update messages are sent rather than invalidations. This allows multiple copies of modified data to exist in the system.

The only difference in the basic implementation of the two protocols is that the update protocol has an extra entry in the directory line. This extra entry is used to indicate when only a single cache holds a copy of the data. It is impossible to determine if only a single cache holds a copy of the data without it, as other directory entries only indicate that a node holds a copy, not how many caches within the node hold a copy. This was a necessary feature to enable acknowledgements to be coordinated correctly within the node.

As with the full-map invalidation protocol, care must be taken to ensure that protocol operations are directed at all of the appropriate nodes and that multiple responses are grouped together before being passed to the lower level. The same overlapping and timing problems that affected the invalidation protocol also affect the update protocol.
Other Network Interface Implementation Issues

To overcome the problems of overlapping requests, lists are maintained in the network interface objects to record the outstanding requests that have originated at, arrived at or have been sent by the network interface. These records include the unique message identifier attached to each request and the memory block being accessed. Further requests for data can then be checked against these records and forced to retry if a request involving the same memory block is still outstanding. This system of forcing retries enables the outstanding request to be completed and coherence to be maintained, before allowing a subsequent request to be issued.

The problem with this mechanism is that a directory entry could have been updated before the retry was issued. Care must be taken to ensure that the directory is returned to the state that it was in before the retried request was issued. This state will be correct as no further requests for data involving that directory entry can be outstanding since they would have been forced to retry before changing the directory entry. The mechanism has the possibility to deadlock, with two requests being constantly forced to retry by different network interface entities, but the chance of this happening is very small due to the non uniform delays associated with the interconnection networks and buses.

The work carried out for this project was not intended to be an exercise in protocol design, which is extremely complicated and very difficult, especially when trying to ensure that the protocol is always correct. The protocols were developed to enable (clustered) distributed shared-memory multiprocessors to be modelled, and they have produced correct results for most of the cases tested. No formal checking has been performed for the upper level protocols to guarantee that coherence is maintained in all possible cases. The results obtained for the experiments carried out in Chapter 8 all executed correctly with coherence being maintained, but this is by no means a guarantee that the protocols are completely correct.
The final implementation issue regarding the upper level protocols is that they are currently compatible with only two lower level protocols. The full-map invalidation protocol supports the Berkeley protocol at the lower level and the full-map update protocol supports the Firefly protocol. The extra work required to convert all of the protocol code to support the network interface entities and an upper level protocol would not prove anything new about the flexibility of the model developed.

This completes the discussion of the implementation of the major entities of the multiprocessor simulation. The only entity not discussed so far is the upper level network, which is a special entity discussed in the next section. The focus of the next section is on how all of these entities are linked together to form a complete system simulation.

6.7 Complete Systems

The previous sections of this chapter have described the implementation of the entities that can be used to construct a simulation of various different computer systems. This section describes how these entities can be connected together to form a complete system. The implementation of the individual entities involves writing code, in C++, that specifies the behaviour of the entity. When constructing the system from a set of entities, no further behavioural code needs to be written. The EDL description of the architecture is used to specify the configuration of the entities to form a system.

Firstly, an EDL description was written for each of the basic entities in the system. The various constructs of EDL, for example links and composite entities (entities that are made up of other entities and links), can then be used to join these basic entities together in the desired manner. The composite entities are used to specify any hierarchy within the architecture, allowing the architecture to be constructed in top-down (entities becoming more and more detailed at each level) or bottom-up (entities becoming more and more abstract at each level). This hierarchy also enables the
display of the design to be simplified and tailored to show the particular parts of the architecture that are of interest. The higher level composite entities can also have their own behaviour code which results in progressively more abstract versions of the simulation, allowing it to be simplified and therefore execute more quickly. However, this extra speed usually comes at the cost of accuracy (see work performed by Williams [Wil99] for a more detailed discussion of hierarchical modelling, its benefits and problems).

The simple single processor system outlined in Section 5.2.4 (and used in a selection of the experiments presented in Section 8.2), was constructed from a processor entity, a number of caches entities and a memory entity. A composite entity was created consisting of the processor and cache entities, which was then connected to the memory entity. The composite node structure was used to reduce the number of changes involved when adding or removing caches, as the memory was connected to the node and not the lowest level cache. The EDL for this simple system can be found in the EDL file shown in Appendix B.1. Figure 6.13 shows a HASE representation of a single processor system with two levels of cache.

![Figure 6.13: HASE representation of a single processor system](image)

The multiple processor systems are constructed in a different manner, using HASE templates. It is possible to construct a multiprocessor system using standard EDL constructs to create the processing node composite entities; these can then be linked to a memory through a bus entity. However, this method of constructing a multiprocessor limits its flexibility as for example, adding more processing nodes
requires extra ports to be added to the bus and extra nodes to be linked to these new ports. This becomes even more cumbersome when clustered systems are modelled and the number of processors per cluster and the number of clusters need to be changed. A different system was required to allow extra nodes to be added with a minimum of effort, ideally by changing a single integer value that represented the number of nodes.

The solution was to develop new HASE templates (see Section 7.3 for a detailed description of implementation of HASE templates). HASE templates construct complex systems from a supplied set of entities and a set of parameters values. For example, the Solaris version of HASE contains three different mesh templates (1D, 2D, and 3D) that construct the appropriate mesh using a supplied entity as the network node and a user specified set of dimensions. These templates also have extra parameters that are used by HASE to control the mesh construction, for example, whether to link the edge ports together and the number of links between each entity.

To allow the multiprocessors discussed in Chapter 5 to be created, a set of templates was created in HASE (one for each type of multiprocessor architecture).

The first of these templates allows simple bus-based multiprocessors that contain a single memory entity to be constructed. The template takes a node entity, bus entity and memory entity as parameters, as well as the number of nodes. The appropriate number of ports are added to the bus and the correct number of nodes are linked to them. Finally, two more ports are added to the bus and the memory connected to them. Figure 6.14 shows the HASE representation of such a bus-based multiprocessor with four nodes. The experiments performed in Section 8.3 on bus-based multiprocessors use this template with nodes constructed from a processor entity and a cache entity.
The second template allows a number of nodes to be connected to multiple memory entities using an arbitrary interconnection network. The basic mechanism used to construct the system is similar to that used in the bus-based multiprocessor template; the main difference is in the construction of the interconnection network.

The interconnection network entity can be supplied to the template in the same manner as that used for node and memory entities, with the entity being created by the designer and having EDL and behavioural descriptions. HASE takes the supplied network entity and adds the correct number of input and output ports. Although this gives the designer the flexibility to design and evaluate any type of network, creating a network entity that is composed of other entities (i.e. more than a "black box") is very difficult as the number of ports is not always fixed, and the I/O ports do not exist until the design is loaded into HASE. This takes away the hierarchical nature of HASE and its ability to animate the operation of the network (as there is nothing to animate).

To overcome these problems special keywords are included in the template which allows standard networks to be used. If one of these keywords is entered as the interconnection network entity, HASE constructs the appropriate network rather than looking for an entity created in the EDL description. HASE currently supports two networks, Crossbar and Omega, but there is no restriction on including more in the
template. The networks constructed can be displayed at a lower level to illustrate the entities and connections that form the network. The network is also capable of adjusting to the number of nodes currently attached. These two features were unattainable with standard EDL, where it would be possible to construct a network that changed according to the number of nodes, or one that could be displayed at the lower level, but not one that was capable of both.

The entities used to construct the networks, for example switches, can use either a HASE supplied behaviour or they can be redefined by the designer by supplying a different hase file (hase files are used to specify the behaviour of entities).

The disadvantage of this type of template network is that the designer is restricted to the networks that have been included; adding more means extending the source code of the HASE templates.

Only two networks were implemented as the model is not intended to be exhaustive. Two networks illustrate that different networks can be created and offer some flexibility for experiments and demonstrate the advantages of using this approach over standard EDL definitions. Figures 6.15 shows the HASE representation of a multiple memory multiprocessor.

The third and final multiprocessor template allows a (clustered) distributed shared-memory multiprocessor to be constructed easily. This template works in the same manner as the multiple memory template, the only difference being that all the nodes connected to the system are the same (unlike the multiple memory system in which half are memory entities and half are processing nodes). The interconnection network is specified in the same way, using either a created network or a HASE supplied network template. To create a distributed shared-memory system the node entity supplied is a relatively simple composite entity with a processor, memory and bus. In contrast, to construct a clustered system, a complete shared-memory multiprocessor is supplied as the node entity (possibly created using a template).
To enable the bus-based shared-memory multiprocessor template described earlier to be used, extensions were required to enable the network interface entity to be included. Figure 6.16 shows a HASE clustered distributed shared-memory multiprocessor, with one node expanded to show the processors, caches, memory, bus and network interface.
Figure 6.16: Clustered distributed shared-memory multiprocessor using a crossbar network

This completes the description of the various templates that have been created to enable multiprocessor systems to be created. The next chapter describes the extensions to HASE that were required to support these multiprocessor models.
Chapter 7

HASE Extensions

This chapter describes the more significant changes to the HASE system that were required to support fully the multiprocessor models described. Many extensions and alterations have been carried out on the HASE system over the course of the project, most of which have been relatively small. By far the biggest change, requiring much work, was the conversion of HASE to enable it to run on a Microsoft Windows PC; at this point many of the minor changes were also performed, for example the ability to send an array down a link and animate it.

The more significant changes that are described here include the ability to use an integer parameter to determine the size of an array parameter, the inclusion of parameters that can specify files to be linked at compile-time, new templates, and a new experiment control mechanism.

7.1 Parameter Dependent Arrays

Early in the design of the cache and memory entities it was realised that a mechanism would be needed to allow the size of array parameters to be dependent on an integer parameter. This would allow the size of arrays to be dynamic and specified at run
time; it would also enable the experiment control facilities to control the size of arrays for different simulation runs.

To include this mechanism into HASE required significant changes to the simulation code that was generated for several of the parameters, including structures and arrays. Changes were also required to the internal architecture structures to enable HASE to detect which integer parameters affected the size of arrays.

This feature became an integral part of the simulation, being used in every entity of the simulation. The obvious use was to allow the size of the cache and main memory storage arrays to depend on an integer parameter. It was used with the bus width parameters, which are used to specify the amount of data that can be sent between two entities. The data being sent is included in an array, which is then sent to another entity and the size of the array used to hold the data is dependent on the appropriate bus width parameter.

7.2 Linker Parameters

The next major extension allowed parameters to control which files were included in the simulation at compile-time. This was used to overcome the problem of changing cache coherence protocols. It was not suitable to have a large behaviour file for each entity that required coherence protocol code or to use conditional statements to select between the various pieces of code, depending on the value of the parameter. This would make coding very difficult and it would be awkward to implement new protocols with the protocol code distributed across multiple large files.

It was not originally envisaged that the parameter system within HASE would be used for the purpose of changing between complicated design alternatives; it was designed for two other purposes. The first is to assign a data structure to an entity, the data structure can be complicated, but it is static and does not change between different simulation runs. The second is to allow the design to be changed easily;
parameters can control memory access time, for example. However, these variable
dparameters are quite simple, for example integers, floating point numbers or
erenumerated types used to represent simple architectural features or parameters.

For the multiprocessor model described here a new system requirement was to be
able to change complex architectural features using a simple parameter, for example
an enumerated type representing the available protocols. The basic mechanism of
including different protocols has already been discussed in some detail in Section
6.3.1; however it focused on the coherence protocol objects and what each protocol
had to implement in order to produce a working system. Here the focus is on the
changes that were required to support the swapping of these protocol objects from a
HASE system perspective.

In order to describe this, a description of how HASE constructs a simulation
executable and how the parameters affect the simulation needs to be presented.

HASE uses three types of file to specify a simulation: EDL, EL and hase. The EL
file determines the layout of the architecture in the HASE display window, not the
simulation. The EDL file specifies the structure of the architecture while the hase
files specify the behaviour of the each of the entities.

Figure 7.1 shows the process used by HASE to convert the EDL and hase files into
an executable simulation.

The EDL file specifies the entities to be used to construct the system to be simulated,
which tells HASE which behaviour files are required for the simulation. The
relevant hase files are then converted into C++ files, as well as generating a project
C++ and header file. These files can then be compiled to form the simulation
executable.
The hase files contain several sections that are dealt with differently by HASE during the conversion process:

- `$class_includes` - specifies any header files required by the entity.
- `$class_decls` - declares any extra variables and functions used by the entity.
- `$class_defs` - contain the definitions of the extra functions declared in the `$class_decls` section.
- `$body` - contains the behavioural code of the entity and is called, for each entity, at the start of the simulation.
- `$startup` - code to be called before an entity starts executing.
- `$report` - code to be called after the entity has completed.

The project C++ file contains the initialisation code used to construct and setup the simulation. The project header file contains the definitions of all the parameters and classes used by the simulation. Both of these files are constructed using the project EDL file and the `$class_decls` and `$class_defs` section of the hase files.
A makefile is also generated by HASE which is used by the compiler to combine all the C++ and header files created by HASE into a simulation executable. This executable requires one input file, the architecture parameter values file, which is generated by HASE. This file is read at the start of every simulation and sets the values of all the parameters specified in the EDL file.

To allow different protocol objects to be included in the simulation this process had to be modified slightly. Each of the protocols was placed in an appropriately named file. The name of the file was the same as the value used in the protocol parameter, for example, the value used for the Write-Once protocol is WriteOnce, which means that the protocol objects should be coded in files called WriteOnce.cpp and WriteOnce.h. HASE can then indicate to the compiler, via the makefile, which extra files are to be included in the simulation by looking at the value of the coherence protocol parameter. The problem is how to indicate to HASE that the coherence protocol enumerated parameter is different from any other enumerated parameter.

The first method considered was to define a completely new parameter in HASE that was used for the purpose of including different files at compile-time. However, this would create a lot of work for what is basically an enumerated parameter, and was therefore considered unsuitable. The second approach considered was to extend the EDL definition of enumerated types to include a mechanism to allow this new functionality to be identified. Although this approach would work, it was decided against using it because of the work involved in extending the EDL parser and the internal structures to indicate the change of status of the enumerated parameter. The approach actually taken was to add a new section, $class_linker, to the hase files that specifies the parameters that are to be used to include different code files. These parameters can then be identified when HASE performs the generation process and the makefile created accordingly. This approach was simple to implement and is easy to use, requiring only minor changes to the HASE system.
This mechanism was used for many different parameters used by the entities in the various multiprocessor models, not just coherence protocols, for example, the processor entity specifies the synchronisation primitives and memory consistency models and the bus entity uses these to implement the bus arbitration scheme.

7.3 New Templates

The templates that existed in HASE were not sufficient to specify the multiprocessor models used in this project. New templates were therefore required to allow the designer to construct a variety of different multiprocessor architectures easily, using a small set of basic entities.

These new templates have already been described in Section 6.7. However that section considered how the templates are used to construct the architecture, not their effects on the HASE system.

To implement a new template there are two main areas that need to be addressed. The first is the EDL parser; it must be updated to allow multiprocessor systems to be described, and new types of entity are therefore needed to describe these complex systems, which allow the various parameters of the different multiprocessors to be specified. Figure 7.2 shows the new EDL entity created for a bus-based multiprocessor; examples of all the multiprocessor templates can be found in the EDL file in Appendix B.1.

The second area to be addressed is the creation of new entity objects. Each different type of entity in HASE has its own object, therefore for each new multiprocessor template created, a new type of object is also required. The implementation of these objects tells the HASE system how to construct the entity and how it relates to the other entity objects used by HASE.
The modular nature of HASE allows new templates to be added with a minimum of effort. This is done by creating a new entity object and extending the parser, allowing the designer to extend HASE to cater for any type of multiprocessor, for example, a new template could be created to represent COMA architectures.

### 7.4 Experiment Control Mechanism

The experiment control facilities were outlined in Section 4.3.6. They are used to run automatically a series of simulations of a proposed system with a variety of parameter values. The experiment control facilities of the Solaris version of HASE were completely separate from the main HASE system, and as such were limited in their usefulness. The control panel listed all the parameters in the system and allowed the designer to specify the start, final and step value for each. Listing all the parameters for the multiprocessor model developed is not practical, as there are many entities, each with a significant number of parameters. The step values were also limited to positive and negative numbers.

The main problem with the original facilities was their inability to access the HASE internal representation of the architecture. This meant that parameters that could be experimented with using this mechanism were limited to simple parameters, for
example integers, floating-point numbers and enumerated types. The multiprocessor model developed enabled the designer to change other, more complicated parameters, for example coherence protocols, network dimensions and even the entities used to construct the system. To allow experiments to be performed easily and over the complete set of parameters offered by the model, the experiment control facilities needed to be extended to include these more complicated types of parameters.

To accomplish this the experiment control facilities included in the PC version of HASE are fully integrated into the environment. This provides them with access to the internal HASE representation of the architecture, and they can also call HASE routines to regenerate the simulation code if required.

The first change was to the mechanism used to specify the parameters to be included in the experiment. This allowed the designer to select the parameters to be included rather than HASE automatically including them all. This was done by adding a box to the parameter display dialog of each entity for each parameter (see Figure 4.7). Only the selected parameters are then included in the experiment control panel.

The next changes to the experiment mechanism were implemented to provide the designer with greater control over the parameter values used in experiments. The original experiment facilities allowed the designer to specify step values of positive or negative integers. This value was then added to the current parameter value to obtain the new value before running the simulation. In the new version the designer is allowed to include multiplication and division, as well as specifying a specific sequence of values that the parameter should be assigned. A list of values option is required, as not all sequences can be easily expressed by simple mathematical expressions.

The designer is also allowed to group parameters together so that they change value at the same time. The experiment mechanism automatically runs simulations for all combinations of the parameter values specified. The grouping option allows the
designer to reduce the number of simulations carried out by tying a number of parameters together, so that when one parameter is updated so are all the others in that group. This would result in a selection of the combinations being performed, so reducing the number of simulations run.

The final new option included in the experiment control facilities is a box that enables the designer to change all of the parameters in the system that have the same name. For example, the multiprocessor system being studied might have many caches and to change the size of them all in an experiment would require that each of them be included in the experiment control panel and grouped together. This could result in many parameters appearing in the experiment control panel and time being wasted in selecting them all and setting their initial, final and step values. The mechanism allows the designer to select the size parameter of one of the caches and then select the “all” box in the experiment control panel. This changes all of the cache sizes in the system whenever the one selected is changed.

The final implementation issue regarding the experiment control facilities was to implement a mechanism to recompile or reconstruct the architecture. This is needed as some of the parameters of the model developed require the simulation to be recompiled before it is run, for example, coherence protocols, synchronisation primitives and memory consistency models, and some require the architecture to be reconstructed, for example those associated with HASE templates, such as number of nodes. This was straightforward to implement in the new experiment control facilities as they are fully integrated into HASE and have complete access to the architecture representation and the internal HASE functions.

7.5 Other Extensions

There were many other extensions and improvements that were made to HASE when converting it to run on a Microsoft Windows PC, and some of these were illustrated in Section 4.3.5, for example the new animator and timing diagram viewer.
Most of the changes made were "behind the scenes" and are not immediately obvious, such as the ability to send arrays down links, to display nested structures in the HASE main display, to construct a complete project using the graphical interface, the simple procedure to allow the designer to change all instances of a parameter in a system, for example cache size, the inclusion of arbitrary length integers and completely new drawing routines to improve efficiency of animation. All of these were necessary to create a complete environment to support the multiprocessor simulation model designed and implemented.

The next chapter presents a selection of experiments that have been performed to illustrate the capabilities of the multiprocessor model and framework. The final chapter, Chapter 9, includes a discussion of how this environment and simulation model compare to those discussed in Chapter 3.
Chapter 8

Simulation Experiments

This chapter presents the results from running a variety of experiments using the multiprocessor model described in Chapter 6. The basic approach used for running these experiments is outlined first, followed by a discussion of each of the experiments, starting with the simpler systems and increasing in complexity up to clustered distributed shared-memory multiprocessors.

8.1 Experimental Methodology

The experiments were performed using the experiment control facilities provided by HASE, which allowed the experiment to be set up and then run automatically without further interference. A detailed discussion of these facilities can be found in Sections 4.3.6 and 7.4.

When creating an experiment, there are several questions that need to be addressed:

- What is the experiment attempting to demonstrate?
- What parameters are involved in demonstrating this?
- What range of values should these parameters take?
- What values should other parameters take?
Each of the experiments described in this chapter will address these four questions as well as presenting and explaining the results obtained. Each of the experiments focusses on a small number of parameters; all other parameters of the system have an impact on the overall results but are constant within the experiment. Some of the more interesting of these parameters will be mentioned, but most of them are of minimum interest for the experiment being performed. A complete list of the values used for all the constant parameters for each experiment can be found in Appendix D.

The experiments were performed on a variety of systems containing different numbers of processors and different configurations. The first experiments described were performed on simple single processor systems, later experiments used bus-based shared-memory multiprocessors and the final experiments were carried out on clustered distributed shared-memory multiprocessors.

All of the experiments assessed system performance for the \textit{lu} program from the SPLASH-2 suite [Woo95] (see Section 6.2 for a detailed discussion of this application). The size of the matrix used in all experiments was 128 x 128, which allowed enough operations to be performed to give accurate and true answers, but was small enough to keep simulation execution times reasonable. The matrix block size used varied depending on the experiment; the dimensions used will be mentioned in the discussion of each experiment.

8.2 Single Processor System Experiments

The first three experiments carried out were based around a simple single processor system. This basic system is illustrated in Figure 8.1. This arrangement was used for the first two experiments discussed, and the third single processor experiment was performed on a system with an extra cache entity inserted between the existing cache entity and the memory entity.
These experiments were performed to demonstrate the flexibility of the cache entity created. This can be shown effectively using a single processor system, without the need for a complex multiprocessor system. The next sections discuss the three experiments that were carried out using this simple system.

8.2.1 Cache Associativity and Block Replacement Policy

The first experiment was to investigate how the associativity of the cache affected its performance. A range of cache associativities was selected to investigate the effect on system performance, starting with direct-mapped and finishing with fully-associative, with 2-way, 4-way and 8-way caches also being studied. The associativity parameter of the cache was therefore assigned the following values: 1 (direct-mapped), 2, 4, 8 and 0 (fully-associative).

As associativity increases, the policy used to decide which cache block to replace when the set is full becomes increasingly important. Some of the blocks are likely to be accessed again quickly, so removing these from the cache could seriously degrade system performance. To investigate the impact of replacement policies, the simulations were repeated for three different replacement policies: random, LRU and round robin.
This addresses the first three questions highlighted in Section 8.1; the final question is the values of all the other parameters in the system. A complete list of these parameter values can be found in Appendix D.1.1. Particular parameters to note are:

- The size of the cache (512 lines) with a block size of 4 (i.e. cache size of 8 KB).
- The cache is copy-back and uses a write-allocate policy.
- The Read and Write delays are 1 cycle for the cache and 5 cycles for the memory.
- The bus width between the cache and memory is 4 words, enabling a complete cache line to be transferred in one cycle.
- The matrix dimension is 128 x 128 with 16 x 16 blocks.

The results obtained from the simulations performed using these parameter values are presented in Graph 8.1. A table of the data collected from this experiment is included in Appendix D.1.2.

Graph 8.1: System performance using different cache associativites for different replacement policies
The graph shows the performance of the simple single processor system with different cache associativities for different replacement policies. The simulated time is the time predicted by the simulation for the modelled system to execute the $lu$ program for the given matrix dimensions. This is different from the simulation execution time, which is the actual time taken to perform the simulation; this measurement will be used in later experiments (see Section 8.3.4).

This simple experiment shows the value of performing simulations to assess the impact of design alternatives. The system is expected to perform better the greater the cache associativity, especially as the cost of the cache increases. This is because there are more possible places in a cache in which to place the requested data, reducing the possibility that data that is required soon will be removed. For example, a direct-mapped cache has no choice when selecting data to be removed as the new data can only be placed in one cache line, however in a fully-associative cache the new data can be placed anywhere.

The graph shows that for this particular application it is not always the case that increasing associativity improves system performance. For example, for LRU replacement policy, increasing from a direct-mapped cache to 2-way set associative cache significantly degrades performance (by 6.5%). Similarly for the round robin replacement policy, increasing from 2-way to 4-way and from 4-way to 8-way also causes a reduction in performance (by 2.1% and 4.7% respectively). Note that for a random replacement policy the expected increase in performance is observed for an increase in associativity.

The loss of performance for the parameter combinations highlighted is caused by the memory access pattern of the application being studied. The particular sequence of accesses used by the application is in conflict with the replacement policy and associativity used. For example, when LRU replacement is used, the application would frequently use the least recently used data, however it is the first piece of data removed by the replacement policy, increasing the number of cache misses and
causing the system performance to worsen. Table 8.1 shows the hit rates for each of the parameter combinations.

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Replacement Policy</th>
<th>Hit Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct-Mapped</td>
<td>Random</td>
<td>85.73</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>85.73</td>
</tr>
<tr>
<td></td>
<td>Round Robin</td>
<td>85.73</td>
</tr>
<tr>
<td>2-Way</td>
<td>Random</td>
<td>89.18</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>81.43</td>
</tr>
<tr>
<td></td>
<td>Round Robin</td>
<td>89.09</td>
</tr>
<tr>
<td>4-Way</td>
<td>Random</td>
<td>89.88</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>89.89</td>
</tr>
<tr>
<td></td>
<td>Round Robin</td>
<td>87.53</td>
</tr>
<tr>
<td>8-Way</td>
<td>Random</td>
<td>90.19</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>90.18</td>
</tr>
<tr>
<td></td>
<td>Round Robin</td>
<td>83.94</td>
</tr>
<tr>
<td>Fully-Associative</td>
<td>Random</td>
<td>98.00</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>96.88</td>
</tr>
<tr>
<td></td>
<td>Round Robin</td>
<td>98.36</td>
</tr>
</tbody>
</table>

Table 8.1: Cache hit rates for different associativities and replacement policies

However, hit rate is not the only factor in deciding the performance of the system. This is illustrated by the hit rates of the 4-way LRU and random caches (89.89% and 89.88% respectively) with simulated times of 19422218 and 19246059 respectively. Therefore LRU has a 0.01% higher hit rate but a worse performance. This is caused by more of the lines selected for replacement by the LRU policy being in a modified state, i.e. they need to be written back to memory.

Both of these (the reduced hit rate and the increase in copy-backs) contribute to the reduction in performance for the parameter combinations highlighted earlier.
The results of this experiment showed that, in general, the performance of the system improves as the associativity of the cache is increased. However, there are several scenarios in which this is not the case, demonstrating the need for designers to be able to perform this type of experiment, even with well understood design parameters.

8.2.2 Cache Write and Allocation Policies

The second experiment carried out also used the simple single processor system with one cache and main memory. This experiment was designed to investigate the impact of the write and allocation policies. The write policy determines whether a write stops at the cache or continues onto main memory. There are two options, write-through (all writes are sent to memory) and copy-back (memory is only updated when the modified line is removed from the cache). The allocation policy determines what actions are to be performed on a write miss. Again there are two options, no write allocate (update only the data in memory) and write allocate (update the data in memory and return the memory block to the cache). This gives a total of four possible combinations to be investigated. Only two of these are generally used, write-through combined with no write allocate, and copy-back combined with write allocate. To illustrate the impact of these two combinations a series of simulations was run for each combination with different sizes of cache.

This addresses the first three questions highlighted in Section 8.1; the final question is the values of all the other parameters in the system. A complete list of these parameter values can be found in Appendix D.2.1. Particular parameters to note are:

- The associativity of the cache (direct-mapped) and a block size of 4.
- The Read and Write delays are 1 cycle for the cache and 5 cycles for the memory.
- The bus width between the cache and memory is 4 words, enabling a complete cache line to be transferred in one cycle.
- The matrix dimension is 128 x 128 with 16 x 16 blocks.
The results obtained from the simulations performed using these parameter values are presented in Graph 8.2. A table of the data collected from this experiment is included in Appendix D.2.2.

Graph 8.2: System performance using different write and allocation policies for various sizes of cache

The graph shows that for smaller caches the write-through/no write allocate strategy performs better than the copy-back/write allocate strategy. For larger cache sizes (8 KB and above) the copy-back cache performs better.

To obtain better performance, copy-back caches rely on the cache line that contains the modified data being rapidly written to again. This reduces the number of writes that have to be performed in main memory as the copy-back policy only updates the cache, effectively combining writes together to form a single write that is passed back to memory in one copy-back operation. Similarly, the write allocate policy
relies on data retrieved on a write miss being rapidly used again. If it is, the data will already be in the cache, so reducing the number of memory accesses. However if it is not accessed again, the time taken to transfer to the data from memory to the cache is wasted and system performance will suffer.

For small caches the chance of the data held in the cache being replaced is greater than for larger caches. Therefore the data modified in the cache or retrieved from memory on a write miss has a shorter length of time in which to be used before it is replaced. If it is replaced before any performance benefit can be gained, the system performance will suffer from the extra operations that are involved in the copy-back and write allocate policies. For larger caches there is a greater chance of this data being accessed again before it is replaced, allowing the system to benefit from the copy-back and write allocate policies.

The graphs flatten out for the larger cache sizes as all the data used by the application fits in the cache. The write-through policy performs worse in this case as all the writes still have to be passed onto main memory, whereas the copy-back cache only writes to the cache and does not have to perform any copy-backs.

The results of this experiment showed that the write and allocation policies have an impact on the performance of the system. However, deciding on which policies to use depends on other factors, for example the size of the cache. The experiment shows that the decisions designers have to make are not always straightforward and they could have implications on other parts of the design. Here the size of the cache has a significant impact on the choice of policies to use, again demonstrating the importance of being able to evaluate many different design alternatives through system simulation.

8.2.3 Multiple Levels of Cache

This experiment uses a single processor system, as in the previous two experiments; however an extra cache is included between the existing cache and the memory. It is
designed to investigate the impact of including more than one cache in the memory hierarchy. This approach to memory hierarchy construction is used because the components used to construct faster caches are expensive, so only smaller caches are constructed from these components to minimise cost. Larger caches are constructed from slower, cheaper components (which are still faster than main memory) and placed between the smaller, faster caches and memory. This allows a hierarchy of components to be created, with slower components being placed further from the processor. This experiment investigates the performance of a system with two levels of cache.

The experiment studies a variety of different configurations for a slower second level cache. The configuration of the faster first level cache is fixed. The second level caches studied vary in their access times (between 2 and 8 cycles) and size (1 to 256 KB). A complete list of the other parameter values can be found in Appendix D.3.1. Particular parameters to note are:

- The size of the first level cache is 512 bytes.
- The associativity of both the caches is direct-mapped and both have a block size of 4.
- Both caches are copy-back and use the write allocate policy.
- The Read and Write delays are 1 cycle for the first level cache and 10 cycles for the memory.
- The bus width between the caches and memory is 4 words enabling a complete cache line to be transferred in one cycle.
- The matrix dimension is 128 x 128 with 16 x 16 blocks.

The results obtained from the simulations performed using these parameter values are presented in Graph 8.3. A table of the data collected from this experiment is included in Appendix D.3.2.
The results presented in the graph show the expected performance benefits obtained by increasing the size of the second level cache. The first interesting point to observe from this graph is that it is not always advantageous to include a second level cache. The configurations of second level cache which cause the system performance to worsen are above the line used to indicate the performance of the system with no second level cache, for example, a 1 KB cache with an access time of 8 cycles. If the level 2 cache is not large enough or fast enough it can cause system performance to degrade due to the increase in the number of messages required to transfer data into and out of the cache; this increase outweighs any advantage obtained from reducing the number of memory requests.

The second piece of information that can be obtained from the graph is how the different speeds of cache compare with each other, allowing cost/performance comparisons to be carried out. For example, a 256 KB cache with an 8 cycle access time performs better than a 16KB cache with a 6 cycle access time (by 1.7%), and
similarly a 4 KB cache with a 4 cycle access time performs better than a 1 KB cache with a 2 cycle access time (by 2.1%). Therefore a particular performance figure can be decided upon and the appropriate cache sizes and speeds that would meet this ideal determined from the graph. The cost of each cache can then be calculated and the cheapest cache used. These calculations may show that it is more cost effective to construct a larger but slower cache than a smaller, faster one.

This type of experiment allows the designer to test different cache design alternatives, allowing the most effective design (in terms of cost and performance) to be selected. It also illustrates that caches do not always improve performance.

8.3 Bus-Based Multiprocessor System

The three previous experiments showed the flexibility of the cache entity by performing a series of experiments on a single processor system. This simple system is not sufficient for demonstrating all the features of the model, in particular those relating to multiprocessor systems, for example, cache coherence protocols and synchronisation primitives. The experiments presented in this section aim to show the flexibility of the model when designing and evaluating multiprocessor systems. To perform these experiments, a bus-based shared-memory multiprocessor system was constructed. Figure 8.2 illustrates an example of such a multiprocessor system with eight nodes, as used in the experiments in this section (unless otherwise stated).

8.3.1 Cache Coherence Protocol

The first multiprocessor experiment focusses on the protocol used to maintain coherence in a shared-memory multiprocessor. The cache coherence protocol is a central part of the shared-memory multiprocessor and it is therefore critical that it maintains coherency in an efficient manner. The experiment is intended to demonstrate that the choice of protocol can have a serious impact on the overall performance of the system.
The main parameter of interest is the coherence protocol parameter of the cache, bus and memory entities. The experiment performs the same set of simulations for all nine of the protocols described in Section 6.3.1. Each of the protocols is studied over a range of different cache sizes (1, 8, 64 and 512 KB) to enable the efficiency of the various protocols to be observed under different conditions. A complete list of the values of the other parameters can be found in Appendix D.4.1. Parameters of particular note are:

- The associativity the caches is direct-mapped and they have a block size of 4.
- The caches are copy-back and use the write allocate policy.
- The Read and Write delays are 1 cycle for the caches, 5 cycles for the memory, and the bus has a cycle time of 2.
- The bus width between the caches, bus and memory is 4 words, enabling a complete cache line to be transferred in one cycle.
- The matrix dimension is 128 x 128 with 16 x 16 blocks.

The results obtained from the simulations performed using these parameter values are presented in Graph 8.4. A table of the data collected from this experiment is included in Appendix D.4.2.

**Graph 8.4:** System performance for different coherence protocols and cache sizes
The results presented in Graph 8.4 illustrate that the choice of coherence protocol has a significant impact on the performance of the system. The first item to note is that for very small caches the choice of protocol is not that important, as they all result in similar performances. This is because it is the size of cache that is the dominant factor in the overall performance and not the coherence protocol. With small caches the chance of any sharing occurring is reduced. This is because the lu program operates on small sections of the matrix in turn (in this case 16 x 16 element blocks) and, as there are 64 of these blocks in the matrix, most blocks will be replaced in the cache before they are used again. This lack of sharing therefore means that the coherence protocol does not have a significant impact on performance.

For small caches the update protocols (Firefly and Dragon) perform better than the majority of the invalidate protocols. However, as the caches increase in size, the performance of the invalidation protocols improves and eventually overtakes the performance of the update protocols. The loss in performance of the update protocols is caused by the number of updates that are required to keep all the data coherent. As the caches become increasingly large, the whole matrix fits in the cache, which results in the need for many update operations (probably one for each write operation) to maintain coherence.

The classical approach performs poorest of all the protocols, due to all write operations being passed to the bus, resulting in bus saturation. The invalidate protocols all perform similarly for large caches, but for medium size caches the Illinois protocol outperforms the others. This is due to its support for cache-to-cache transfers, ownership and a reduced number of protocol enforced copy-backs.

Experiments such as these enable the designer to evaluate the performance of different coherence protocols. These protocols can be central to the performance of the system and the ability to change between them easily in the model is essential when evaluating possible design alternatives.
8.3.2 Number of Processors in a Bus-Based Multiprocessor

The experiments presented so far have concentrated on changing architectural features and not the actual architecture. This experiment evaluates how the number of processors included in a bus-based shared-memory multiprocessor affects the performance of the system. This will demonstrate the ability of HASE to change the configuration of the underlying architecture based on a simple parameter. The parameter of interest in this experiment is a template parameter, i.e., number of nodes.

The number of nodes on the bus investigated was varied between 1 and 32. All the simulations were performed twice, once with an invalidation protocol (Berkeley) and once with an update protocol (Firefly). This enabled the performance of the two different types of protocol to be evaluated with different numbers of caches in the system.

That outlines the parameters of interest in this experiment. A complete list of the values of the other parameters can be found in Appendix D.5.1. Parameters of particular note are:

- The caches are 16 KB with a direct-mapped associativity and a block size of 4.
- The caches are copy-back and use the write allocate policy.
- The Read and Write delays are 1 cycle for the caches, 5 cycles for the memory, and the bus has a cycle time of 2.
- The bus width between the caches, bus and memory is 4 words, enabling a complete cache line to be transferred in one cycle.
- The matrix dimensions is 128 x 128 with 8 x 8 blocks. Smaller blocks have been used in this experiment to enable enough data parallelism to exist so that all the processors perform a reasonable amount of work.

The results obtained from the simulations performed using these parameter values are presented in Graph 8.5. A table of the data collected from this experiment is included in Appendix D.5.2.
This graph illustrates that performance is not proportional to the number of processors over the range illustrated. By doubling the number of processors from 1 to 2, the time to execute the \textit{lu} program almost halves (43% improvement for Berkeley and 42% for Firefly), and this level of performance improvement continues when 4 processors are added. However, the benefits of adding more processors start to decrease. The worst improvement is from 16 to 32 processors, which offers an improvement of only 4% for Berkeley and less than 1% for Firefly. This demonstrates that increasing the number of processors above 8 would not be very cost-effective.
This loss of improvement in performance is due to several factors. The first factor is the decrease in the amount of work that each of the processors execute. For later iterations of the algorithm in a 32 processor system, many of the processors will perform very little work. The second factor is a sequential section in the \textit{lu} program that is only performed by one of the processors, so the quicker the parallel matrix algorithm section of the code is executed, the more influence the sequential section has on the overall performance. The final and probably most important factor is that with more processors connected to the single shared bus, contention for this limited resource is high. This results in processors having to wait longer for bus access, causing memory requests to take longer to complete.

The other piece of information that can be obtained from this graph is that there is little difference between the two different protocols, although the performance of the update protocol flattens out a little earlier. For example, the Firefly protocol only improves performance by 0.5\% when moving to 32 processors from 16 processors, whereas the Berkeley protocol improves by 4\%. The performance of the Berkeley protocol for 32 processors is 8.5\% better than the Firefly protocol.

This experiment highlights the importance of being able to evaluate different architectural configurations. The ability to change a single parameter provides the designer with a simple mechanism for performing this type of evaluation.

\textbf{8.3.3 Different Processor Speeds}

The third multiprocessor experiment is designed to study the effect of using faster processors to execute the matrix algorithm. This is an important area to be able to evaluate, as there are many different processors that could be used to construct a system. The faster the processor, the more expensive it will be; however faster processors may not translate into a faster multiprocessors, as all requests still have to be dealt with by the same coherence protocol and they will all share the same bus. Therefore by allowing different processors to be evaluated the most cost effective solution can be found.
It is possible to introduce different processor models into the simulations using the template parameter that specifies the processor entity to be used. However this is not the approach that will be used here. As described in Section 6.2.2 the \textit{lu} program is synchronised with the multiprocessor simulation by the introduction of hold instructions in the processor. These holds depend on the instructions that were executed between successive memory requests. To calculate the delays the \textit{lu} program was rewritten in assembly code, enabling the numbers and types of instructions between successive memory requests to be recorded. As described in Section 6.2, the processor entity implemented in the simulation model has a set of parameters associated with it that represent the delay for each type of instruction used by the \textit{lu} program. By altering the values of these parameters, the effects of introducing different processors can be evaluated, for example, setting all of the delays to zero would reduce the time between memory requests to zero, assessing the performance of the complete system with an extremely fast processor.

There are 10 instruction delay parameters associated with the processor entity. To reduce the number of simulations to be performed, these were grouped into 6 categories: integer arithmetic, integer multiply, double precision floating point (DP) arithmetic, DP multiply, DP divide and branches. Each of these categories was assigned three values for this experiment, a minimum, maximum and middle value, for example the three values of the integer multiply delays are 0, 3 and 6. The minimum value for all of the categories is 0 and the maximum value is always double the middle value. The middle values were chosen to reflect the delay of the actual instructions, with a small delay being assigned to integer arithmetic and longer delays assigned to double precision operations. Fifteen experiments were then performed using these values. The first three simulations set all delay parameters to their minimum value, then their maximum value and finally their middle values. For each of the 6 instruction categories, 2 simulations were performed. The first simulation set one instruction category's delay to its minimum value with all other delay parameters set to their middle value; and the second simulation set one instruction category's delay to the maximum value with the remainder set at their
middle value. This was repeated for each instruction category. This enables the impact on performance of each of the instructions to be evaluated.

A complete list of the values of the other parameters can be found in Appendix D.6.1. Parameters of particular note are:

- The caches are 16 KB with a direct-mapped associativity and a block size of 4.
- The caches are copy-back and use the write allocate policy.
- The Read and Write delays are 1 cycle for the caches, 5 cycles for the memory, and the bus has a cycle time of 2.
- The cache coherence protocol used is Illinois.
- The bus width between the caches, bus and memory is 4 words, enabling a complete cache line to be transferred in one cycle.
- The matrix dimensions is 128 x 128 with 16 x 16 blocks.

The results obtained from the simulations performed using these parameter values are presented in Graph 8.6. A table of the data collected from this experiment is included in Appendix D.6.2.

This graph illustrates the instruction categories that have the most influence on the performance of the system. The categories that deviate most from the middle horizontal line are those that have the most significant impact on performance. The first point of note is that the double precision divide instructions, despite having the largest delay associated with them, have the least impact on the overall performance. This means that the frequency of these instructions is relatively low, for example, decreasing the delay from 24 cycles to 12 cycles improves performance by 1%. In contrast to this the integer arithmetic instructions have the lowest delay and yet have a significant impact on performance; this to due to the large amount of integer arithmetic instructions that are executed. Decreasing the arithmetic instruction's delay from 2 cycles to 1 cycle improves performance by 11%.
This sort of information can allow designers to identify the areas of the processor design that should be focussed on. The double precision multiplication instructions are shown to have a significant impact on performance and therefore improvements in the design of the relevant functional units in the processor would improve the performance. The delay of these double precision multiplications is assumed to be 6 cycles, leaving plenty of room for improvements to be made. The integer arithmetic instructions have a similar impact on performance, but their delay is assumed to be 1 cycle, leaving little room for improvement. The designer’s time is therefore better spent focussing on the double precision multiplication functional units.
The second point of interest that can be observed from the graph is that the sum of the individual improvements obtained by setting each of the instructions categories delay to zero does not equal the performance of setting all of the delays to zero at once. The predicted performance, obtained by considering the improvements of the individual categories, is better than the actual performance measured for the system whose instructions are all performed in zero time. This can be explained by considering the extra load that is placed on the bus by the ability of the processor to execute all of the instructions in effectively zero time. Each processor would issue the next memory request as soon as the previous one had completed; this would saturate the bus, causing the average time to satisfy the memory request to increase. Setting all of the delays to zero allows the theoretical maximum performance of a specific architecture to be determined that is independent of the speed of processor, i.e., no matter the speed of the processor used, the system cannot perform any better. At this point changes must be made to the architecture of the multiprocessor if better performance is required.

This experiment illustrates the effect of the processor on the performance of the system. It allows designers to see how their particular multiprocessor will perform as new generations of processor are introduced, and whether a new design is needed to take advantage of these faster processors.

8.3.4 Synchronisation Primitive Implementation

The purpose of this experiment is to illustrate that by implementing more abstract versions of particular architecture features, simulation execution times can be improved at the expense of accuracy. The focus of the experiments is the implementation of synchronisation primitives. As outlined in Section 6.2.3, synchronisation primitives can be easily implemented so that they do not use the simulated shared-memory. If the synchronisation variables are stored in the memory of the host machine, access to them is much faster than through the simulated memory, improving the execution time of the simulation.
To perform this experiment three different implementations of the same synchronisation primitives were created. The first accessed the simulated memory to obtain the value of the synchronisation variables. The second used a variable stored in the memory of the host machine and continually checked the contents of the variable to determine whether it could proceed or not, i.e. it used busy-waiting. The third implementation also used a variable in the memory of the host machine but used interrupts to inform the next processor that it could proceed, not busy-waiting. The other parameter varied was the cache coherence protocol; the three synchronisation primitive implementations were run for each protocol.

A complete list of the values of the other parameters can be found in Appendix D.7.1. Parameters of particular note are:

- The caches are 16 KB with a direct-mapped associativity and a block size of 4.
- The Read and Write delays are 1 cycle for the caches, 5 cycles for the memory, and the bus has a cycle time of 2.
- The matrix dimensions is 128 x 128 with 16 x 16 blocks.

The results obtained from the simulations performed using these parameter values are presented in Graph 8.7. A table of the data collected from this experiment is included in Appendix D.7.2.

The graph is split into four groups. The Berkeley and Firefly simulated groups show the predicted performance of the simulated system using the three different implementations of the synchronisation primitives. The Berkeley and Firefly execution groups show the simulation execution times of the three implementations, i.e., the actual time taken to perform the simulations on a PC.
The three different implementations, shared-memory, private busy-waiting and private interrupts result in almost identical system performance. This may at first seem strange as one of them accesses the simulated shared-memory in order to retrieve the values of the synchronisation variables, which should result in a slower system. However, the spin-locks are only used once by each of the processors (to obtain their unique identifier) and the barriers are only used to separate iterations of the matrix algorithm. When a processor arrives at a barrier it reads the synchronisation variable, causing it to be fetched into the cache. Further reads of this variable are satisfied by the cache, which does not use the bus, and therefore does not affect any other memory requests. While the processors are continually reading the synchronisation variable, other processors are performing actual calculations on the matrix, so most of the synchronisation requests are performed in parallel with matrix calculations. Therefore removing these requests from the system would not have large impact on the predicted performance of the system. The small differences in the simulated times are a result of the synchronisation requests that miss in the cache, i.e., the initial requests, and when the variables are updated (when all the processors have arrived at the barrier).
The error that is introduced by using implementations that do not use the simulated shared-memory is very small. For the Berkeley protocol the error in the simulated time is 0.16% (for both implementations) and for the Firefly protocol it is 0.12% (for both implementations).

The most dramatic feature of Graph 8.7 is the large reduction in simulation execution time for the implementations that use the memory of the host machine. This is due to the dramatically reduced number of events that are processed by the simulation. Each time a processor accesses the simulated shared-memory, many events are generated, and these events are generated for each cycle that the processor is held at the barrier. By removing these events the simulation execution time reduces significantly, as can be seen from the reduction in time in moving from the shared-memory bar to the private busy-waiting bar of the graph (42% reduction for Berkeley and 58% for Firefly).

The further reduction in simulation execution time observed by using the interrupt implementation is a result of the processors performing no instructions while they are waiting for all other processors to arrive at the barrier. This enables the entities that are active to use more of the host machine's processor. This reduces the execution time of the simulation when compared to the shared-memory implementation (by 56% for Berkeley and 66% for Firefly).

This experiment demonstrates that large reductions in simulation execution time can be obtained by sacrificing the accuracy of the simulation. The designer is able to perform trade-offs between the accuracy of the simulation and the time to execute it. The faster simulations could then be used to guide the early design process to remove design alternatives that perform very poorly more quickly, although care must be taken when making design decisions on simulations that are not completely accurate. Reintroduction of architectural features that were abstracted out for simulation purposes may also have an unexpected impact on the performance of the system.
8.4 Clustered Distributed Shared-Memory Multiprocessor System

The previous two sections have presented experiments on single processor or bus-based shared-memory multiprocessor systems. The final section in this chapter will present experiments that have been performed on clustered distributed shared-memory multiprocessor systems that illustrate some of the design options that can be explored using this model. Figure 8.3 illustrates a typical system used in these experiments.

![Figure 8.3: A typical clustered distributed shared-memory multiprocessor system](image)

### 8.4.1 Cache Size in a Clustered System

This first experiment shows how the size of the caches used influences a complete clustered multiprocessor system. It is intended to demonstrate that, in a system with many entities connected over multiple networks, a simple parameter such as the size of the caches can still have a significant impact on the performance of the system.
The size of the caches were varied between 512 bytes and 512 KB and the clustered system used the Berkeley protocol to maintain coherence within the clusters and the Full-Map Invalidation protocol to maintain coherence between clusters.

A complete list of the values of the other parameters can be found in Appendix D.8.1. Parameters of particular note are:

- There are 4 clusters in the system each with 4 processors.
- The interconnection network used at the upper level is a crossbar.
- The Read and Write delays are 1 cycle for the caches, 5 cycles for the memory, and the bus has a cycle time of 2, as does the crossbar network.
- The matrix dimensions is 128 x 128 with 16 x 16 blocks.

The results obtained from the simulations performed using these parameter values are presented in Graph 8.8. A table of the data collected from this experiment is included in Appendix D.8.2.

**Graph 8.8:** Performance of a 4 cluster system (4 processors per cluster) with different cache sizes
Graph 8.8 shows that the size of the caches has a significant impact on the performance of the system. Increasing the size of the cache up to 32 KB dramatically improves the overall performance; however increasing the size of the cache beyond this value is not worthwhile as the increase in performance is very small.

The second point to notice about the performance of the 4 cluster system is that the simulated execution times are significantly lower than those obtained for any of the experiments performed so far. In Section 8.3.2 the effects of the number of processors connected to the bus was investigated and the performance of the 16 processor system was 4183109 simulation time units; here 16 processors performed the matrix calculation in 2515679 simulation time units, an improvement of 40%. This improvement would be even greater if the clustered system had used a matrix block size of 8 x 8 (as did the 16 processors on the bus) instead of the 16 x 16 used here, as this would increase the amount of data parallelism in the code and allow more processors to perform useful work in the later rounds of the algorithm.

This experiment demonstrates the importance of being able to adjust any parameter at any level of the architecture. Even though the caches are low down in the hierarchy they still have a significant impact on the performance of the system.

8.4.2 Clustered Architecture Configurations

The final experiment carried out investigated how the different configurations of cluster affect the performance of the system, i.e., do a few clusters with many processors perform better than many clusters with only a few processors. The second area that this experiment focussed on is how the speed of the interconnection network affected these configurations.

The experiment will consider a multiprocessor system with 32 processors, in which the number of clusters varies between 1 and 32, and the number of processors within
a cluster varies between 32 and 1. The interconnection network used is a crossbar with delays of 2, 10 and 18 cycles.

A complete list of the values of the other parameters can be found in Appendix D.9.1. Parameters of particular note are:

- The caches are 16 KB.
- The Read and Write delays are 1 cycle for the caches, 5 cycles for the memory, and the bus has a cycle time of 2.
- The coherence protocols used are Berkeley and Full-Map Invalidate.
- The matrix dimensions is 128 x 128 with 8 x 8 blocks.

The results obtained from the simulations performed using these parameter values are presented in Graph 8.9. A table of the data collected from this experiment is included in Appendix D.9.2.

Graph 8.9: Performance of different cluster configurations for different interconnection network speeds
The graph illustrates that the \textit{lu} program performs better with a small number of processors per cluster. The main reason for this is that there is very little active sharing of data between the processors, i.e., processors are not actively working on the same area of the matrix at the same time. The sharing pattern of the algorithm is more migratory, with a block only being used by a processor after other processors have finished with it, causing the data to migrate between the caches of the processors that access the block. Smaller numbers of clusters with more processors in the cluster would perform better if the processors in a cluster were sharing data, as sharing this data over the interconnection network (as would happen with more clusters) would be very time consuming. This local sharing would overcome the potential loss of performance that would arise through more processors sharing a bus.

The performance decreases when moving from 16 to 32 clusters; this is because there are too many requests traversing the interconnection network, causing the system to slow down. The best performance was observed for the 16 cluster system as this balances the utilisation of the bus with the number of requests traversing the interconnection network.

The other interesting feature of the graph is the effect of increasing the interconnection network delay on systems with higher numbers of clusters. For faster networks (e.g. a delay of 2), the configuration with 32 clusters performs better than the configuration with 8 clusters (by 2.6%). However, as the delay increases (to 10), the performance advantage gained by using 32 clusters decreases (to 0.8%). Eventually the 8 cluster configuration performs better than the 32 cluster system, for example, an interconnection network delay of 18 results in a performance for the 8 cluster system that is 0.8% better than the 32 cluster system. This is due to the greater amount of network traffic for higher numbers of clusters, therefore the increase in interconnection network delay affects a larger percentage of the messages in these systems. This causes the performance to decrease further for these cluster configurations.
This experiment demonstrates the capability of HASE to explore and evaluate different architecture configurations with a minimum of effort.

The experiments presented in this chapter do not cover all of the possible experiments that could be performed using this multiprocessor model. It would have been impossible to perform an exhaustive set of experiments; the ones carried out illustrate the possibilities that are available using this model. They also demonstrate the importance of simulation as tool for computer systems designers. Many of the experiment results contain anomalies that would have been difficult to predict without simulation, illustrating that it is impossible to always predict the performance of large computer systems, and the impact that changing a single parameter of the architecture can have on performance.
Chapter 9

Conclusions

More powerful computer systems are needed to deal with the large computing problems that are present in today's technological world, such as those that are associated with the computing grand challenges, for example, Quantum Chromodynamics, Ocean Circulation and Weather Modelling. As systems increase in complexity in order to tackle these problems, it becomes very difficult to predict their performance. Consequently there is a need for a system that enables a designer to experiment with different machine designs and explore the trade-offs between different architectural design parameters.

Many simulations of multiprocessors have been created over the years, but these have generally focussed on a limited number of architectural features, for example, the coherence protocol, the number of processors or cache/memory size. This approach limits the options available to a designer when exploring an architecture design; the limited number of parameters also makes it difficult to assess the full impact of a particular design decision under different conditions.

The multiprocessor simulation model created in this work allows a wide range of architectural parameters to be altered, providing the designer with much more
freedom when exploring a design. A framework has also been developed to allow the designer to perform these design alterations with a minimum of effort.

This chapter first summarises the model and framework designed and implemented. This is followed by a comparison of the model and framework to other simulations that have been created. The final section of this chapter details the directions in which future work could extend the system.

9.1 The Multiprocessor Model and Simulation Framework

The model developed is based around a small number of core entities, each having a set of parameters that enable the behaviour to be changed. These entities are also able to select different implementations of key architectural features such as coherence protocols and synchronisation primitives. A variety of different system configurations has been constructed from these core entities using HASE templates, varying from simple single processor systems to clustered distributed shared-memory multiprocessors. These templates also have parameters associated with them, enabling the configuration of the complete system to be varied, for example, the number of nodes and the particular entities used.

In order to take advantage of the features of the model, a simulation framework was developed by extending the HASE environment. These extensions included the implementation of new experiment control facilities and new architectural templates, enabling a designer to explore and evaluate an architecture with a minimum of effort.

To demonstrate the capabilities of the multiprocessor simulation models developed, a series of experiments was performed. These investigated a wide spectrum of architectural features, ranging from simple parameters such as different entity speeds and bus widths, to more complicated features, for example, coherence protocols, synchronisation primitives and system configurations. The results illustrated the advantages of using these parameterised models in conjunction with the simulation
framework, demonstrating the importance of simulation in the design and evaluation of multiprocessor systems.

9.2 Comparison to other Simulation Models and Frameworks

As outlined in Chapter 3, many multiprocessor simulation models and frameworks have been constructed. This section compares these systems to the one created during this project and outlines the advantages and disadvantages of using this approach.

The simulation models discussed in Chapter 3 allowed only a limited number of architectural features to be changed. The simulation developed by Eggers and Katz [Egg88] supported variations in the number of processors (5-12) and the use of either the Berkeley and Firefly protocols. Pai et al [Pai96] allowed different consistency models to be explored on systems with 8 or 16 processors, using copy-back or write-through caches. The simulation developed by Wood et al [Woo93] compared seven different coherence protocols. This restricts the evaluation, as only these architectural features can be changed, for example, a particular coherence protocol may perform better with a specific size of cache or associativity and if the model does not allow these to be investigated, incorrect design decisions could result.

The model created for this project allows a wide range of architectural parameters to be experimented with, ranging from simple delay parameters that adjust the speed of entities, to complex parameters that change the configuration of the architecture, interconnection network topology or coherence protocol. This allows the designer to explore the design space fully and to make more informed decisions about the correct architecture.

The simulation framework developed within HASE provides improved support for multiprocessor modelling. This allows many experiments to be performed automatically and the architecture to be configured with a minimum of effort. There
are many simulation environments that have been developed, for example, SimOS [Ros97] and Ptolemy [Dav99], which provide little or no support for the simulation of multiprocessor systems. PROTEUS [Bre91] is one of the few developed which provides some support for multiprocessor simulation and allows different implementations for the major multiprocessor components to be included. A novel and powerful feature provided by HASE to support multiprocessor modelling is the use of parameterised templates; these allow the architectures to be parameterised, enabling different configurations and implementations to be specified quickly and simply.

The main disadvantage of the HASE simulation model is simulation execution time. The model was developed to be flexible, allowing different alternatives to be explored; however, this results in longer simulation execution times due to the extra code required. The simulation models and frameworks previously developed and outlined in Chapter 3 were created to study the impact of a small number of architectural features, or to study the performance of a particular design, enabling the simulation to be coded in a manner that takes advantage of the fixed state of most of the architecture. This allows the simulation to be implemented in a specific manner, or use special simulation techniques such as binary translation [Wit96] in an attempt to minimise the execution time. Consequently these simulation models take less time to execute.

However, by comparing only execution times, the different degrees of flexibility of the models are not considered. Previous models have sacrificed flexibility in order to improve simulation execution times; in contrast, the model developed in this project focussed on flexibility, resulting in slower simulations. Comparisons between models should therefore take into account the time involved in changing different aspects of the architecture, as well as the simulation execution time.

Although the actual simulations take longer in the model developed here, the time between successive simulations is extremely short as the experiment control facilities update the parameter values (and possibly reconstruct the architecture) before
running the next simulation. In contrast, the models developed elsewhere do not support this simple method of parameter changing and architecture reconfiguration, requiring new models to be coded that implement the different parameter options. Running a series of experiments therefore requires the execution of different implementations in the correct order; this process could be automated, but this would require a new tool to be developed.

In the model developed in this work, not only is it simple to change parameter values and architecture configurations, the model is easily extendable. This can be achieved in several ways. The first is the implementation of new parameter options for existing parameters, for example, adding new coherence protocols. The second is to add a new parameter to an entity and modify its code to reflect the new parameter. The third is to develop a completely new implementation of an entity and substitute it into the simulation.

The other simulation models that have been developed are particularly difficult to extend. PROTEUS [Bre91] supports the incorporation of new component implementations, but this system is one of the few that considers this a useful feature. The specialised implementation of most of the simulation models makes including new parameters, new parameter implementations and new component implementations extremely difficult.

The model developed here approaches the modelling and simulation of multiprocessor systems from a different perspective. The focus of the approach is on flexibility, enabling the designer to explore and evaluate a large number of design possibilities and to extend the model easily to incorporate new ideas.
9.3 Future Directions for the Multiprocessor Model and Simulation Framework

The HASE multiprocessor model and simulation framework developed cover a large range of parameters and parameter values; however there are many others that could be incorporated into the design. This section discusses some possible directions for future work.

The first area that could be extended is the choices available to the designer for the parameters of the entities. This would include the implementation of more protocols, synchronisation primitives, memory consistency models and bus arbitration schemes. It would also include the extension of the template parameters, for example, the inclusion of more interconnection topologies supported by HASE. This would enable more comprehensive experiments to be performed when evaluating different architecture configurations or applications.

The second area in which the model could be extended is the development of different versions of the entities. This could include, for example, the development of a cache entity that supported non-blocking loads and a bus entity capable of dealing with multiple requests simultaneously. It could also include the development of different processor entities that supported different methods of driving the simulation, for example, an instruction emulator capable of interpreting assembly code. New processor models could also be developed that include more advanced processor features, for example, instruction level parallelism.

Thirdly, more applications could be converted from SPLASH-2 [Woo95] and NAS benchmarks [Bai95], as well as other parallel benchmark suites. This would allow evaluation of the architectures designed under different workloads, enabling decisions to be made based on data obtained from a range of different applications.

Another area of future work would be the development of more multiprocessor templates in HASE, for example, a COMA architecture template. These templates
would allow different architecture styles to be evaluated, using the approach developed. Once multiple templates exist, a new high level template could be created that allowed the architecture style, for example, UMA, NUMA or COMA, to be specified as a parameter of the architecture. This would enable the model to vary the architecture style as easily as it could change a delay, protocol or configuration, allowing the experimentation control facilities to automatically perform experiments that change the architecture style as well as other architecture features.

Also the EDL language could be extended to allow designers to specify their own architecture templates. This would significantly increase the complexity of the EDL language and parser, but would allow new templates to be included without the need to change the HASE source code.

Finally, to overcome the problem of longer simulation execution times, the model could be converted to execute on a parallel machine; these machines could be used in two different ways. The first would be to run a complete simulation on each processor of the parallel machine concurrently, with each simulation having a different set of parameter values. This would enable many different alternatives to be evaluated rapidly, effectively in the time it takes to run one simulation. The second method would be to parallelise the multiprocessor simulation model, by either implementing a parallel version of HASE++, or replacing HASE++ with a parallel simulation language (for example, Parsec [Bag98]). This would allow the model to execute using a number of processors, so speeding up the simulation execution time.

Overall, the multiprocessor model developed has demonstrated the value of being able to simulate a wide range of parameters and parameter values, and how this can benefit the designer. The model's design also allows future developments to be incorporated with a minimum of effort, making it a multipurpose design and evaluation tool.
References


232
and the 9th Symposium on Parallel and Distributed Processing, pp. 784-790, 30th March - 3rd April 1998.


Appendix A

A.1 EDL Grammar

Project → PROJECT { Preamble Paramlib Globals EntLib Structure }

Preamble → PREAMBLE { Name Directory Author Version Description }

Name → NAME string

Directory → DIRECTORY string

Author → ε

  | AUTHOR string

Version → ε

  | VERSION float

Description → ε

  | DESCRIPTION Description_List

Description_List → string

  | string , Description_List

ParamLib → PARAMLIB { Param_List }

Param_List → ε

  | Param ; Param_List

Param → ENUM ( identifier , [ Enum_List ] )

  | STRUCT ( identifier , [ Struct_List ] )

  | RANGE ( identifier , integer , integer )

  | INSTR ( identifier , [ Link_List ] , identifier )

  | BIT ( identifier , integer )

  | LINK ( identifier , [ Link_List ] )

  | ARRAY ( identifier , Array_Size , identifier )

Enum_List → identifier

  | identifier : identifier

  | identifier , Enum_List

249
identifier : identifier , Enum_List

Struct_List → R_Param
   | R_Param , Struct_List

Link_List → ( identifier , R_Param )
   | ( identifier , R_Param ) , Link_List

Array_Size → integer
   | identifier

R_Param → RENUM ( identifier , identifier , integer Hide )
   | RSTRUCT ( identifier , identifier Hide )
   | RRANGE ( identifier , identifier , integer Hide )
   | RINSTR ( identifier , identifier Hide )
   | RBIT ( identifier , identifier Hide )
   | RLINK ( identifier , identifier Hide )
   | RARRAY ( identifier , identifier Hide )
   | RINT ( identifier , integer Hide )
   | RFLOAT ( identifier , float Hide )
   | RSTRING ( identifier , string Hide )
   | RH_INT ( identifier , string Hide )

Hide → ε
   | , HIDE

Globals → GLOBALS { R_Param_List }

R_Param_List → ε
   | R_Param ; R_Param_List

EntLib → ENTITYLIB { Ent_List }

Ent_List → ε
   | Entity ; Ent_List
   | Sub_Entity ; Ent_List
   | 1D_Mesh_Entity ; Ent_List
   | 2D_Mesh_Entity ; Ent_List
   | 3D_Mesh_Entity ; Ent_List
   | Bus_Entity ; Ent_List
   | Multiple_Memory_Entity ; Ent_List
Network_Entity ; Ent_List

Entity → ENTITY identifier ( Ent_Desc Params Ports Attributes )

Sub_Entity → COMPENTITY identifier ( Ent_Desc Descendants Params Ports Attributes )

1D_Mesh → MESH1D identifier ( ENTITY_TYPE ( identifier )
NO_ENTITIES ( integer )
NO_LINKS ( integer )
SPACING ( integer )
Wrap Direction Ent_Desc Params )

2D_Mesh → MESH2D identifier ( ENTITY_TYPE ( identifier )
X_SIZE ( integer )
Y_SIZE ( integer )
NO_LINKS ( integer )
X_SPACING ( integer )
Y_SPACING ( integer )
Wrap Ent_Desc Params )

3D_Mesh → MESH3D identifier ( ENTITY_TYPE ( identifier )
X_SIZE ( integer )
Y_SIZE ( integer )
Z_SIZE ( integer )
NO_LINKS ( integer )
X_SPACING ( integer )
Y_SPACING ( integer )
Z_SPACING ( integer )
Wrap Ent_Desc Params )

Bus_Entity → BUSENTITY identifier ( NODE ( identifier )
MEMORY ( identifier )
BUS ( identifier )
NETWORK_INTERFACE ( identifier )
NUMBERNODES ( integer )
Ent_Desc Params Ports Attributes )

Multiple_Memory_Entity → MULTIPLEMEMORY identifier (
NODE (identifier)
MEMORY (identifier)
NETWORK (identifier)
NUMBERNODES (integer)
NUMBERMEMORIES (integer)

Ent_Desc Params Ports Attributes)

Network_Entity → NETWORKENTITY identifier (NODE (identifier)
NETWORK (identifier)
NUMBERNODES (identifier)

Ent_Desc → DESCRIPTION ( )
   | DESCRIPTION (Desc_List)

Params → PARAMS ( )
   | PARAMS (R_Param_List)

Ports → PORTS ( )
   | PORTS (Port_List)

Port_List → Port
   | Port; Port_List

Port → PORT (identifier, identifier, identifier)
   | ARRAYPORT (identifier, identifier, identifier)
   | SUBPORT (identifier, identifier, identifier)

Attributes → ATTRIB (Attrib_List)

Attrib_List → Attrib
   | Attrib; Attrib_List

Attrib → ε

Wrap → ε
   | WRAP (integer)

Direction → ε
   | DIR (integer)

Descendants → DESCENDANT (Child_List ChildLinkList)

Child_List → ε
| CHILD ( identifier, identifier, Attributes );
| CHILD ( identifier, identifier, Attributes ); Child_List

Child_Link_List $\rightarrow \epsilon$

| C_Link ; Child_Link_List

C_Link $\rightarrow$ CLINK ( identifier.identifier[identifier] $\rightarrow$ identifier.identifier [identifier] Width )

Width $\rightarrow \epsilon$

| , integer

Structure $\rightarrow$ STRUCTURE { Structure_List Child_Link_List }

Structure_List $\rightarrow \epsilon$

| Structure_Entity ; Structure_List

Structure_Entity $\rightarrow$ AENTITY identifier identifier ( Ent_Desc Attributes )
Appendix B

This appendix contains the EDL file for the systems simulated. It contains definitions for single processor systems, as well as bus-based shared-memory multiprocessors, multiple common memory multiprocessors and clustered distributed shared-memory multiprocessors.

B.1 EDL Description of Multiprocessor Systems

```plaintext
PROJECT {
    PREAMBLE {
        NAME "Multiprocessor Cache Simulation"
        DIRECTORY "D:\Hase\Projects\Multiprocessor"
        AUTHOR "Paul Coe"
        VERSION 3.2
        DESCRIPTION "EDL for the parameterised multiprocessor"
    }
    PARAMLIB {
        -- Definitions of the enumerated types used by the parameters
        ENUM ( tMemoryAction , [READ,WRITE,COPYBACK,STOP] );
        ENUM ( tCacheState , [RH, RM, WH, WM] );
        ENUM ( tBusState , [BUS_IDLE,BUS_BUSY] );
        ENUM ( tSynchronization , [CentralBarrier,CentralBarrierBW, CentralBarrierOpt] );
        ENUM ( tMemoryConsistency , [Sequential_WeakOrder] );
        ENUM ( tAllocationPolicy , [WRITE_ALLOC,NO_WRITE_ALLOC] );
        ENUM ( tWritePolicy , [COPY_BACK,WRITE_THROUGH] );
        ENUM ( tReplacementPolicy , [RANDOM,LRU,ROUND_ROBIN] );
        ENUM ( tCoherenceProtocol [NoProtocol,Classical,MESI,MOESI, Synapse,Berkeley,Illinois,Firefly,Dragon,WriteOnce, FullMap,FullMapUpdate] );
        ENUM ( tArbitrationScheme , [BUS_ROUND_ROBIN] );

        -- Definition of the cache array structure
        BIT ( tCacheStateBits , 8 );
        ARRAY ( tValueList , BlockSize , int );
        STRUCT ( tCacheLine , [RINT(Valid,0), RINT(Tag,0),
            RARRAY(tValueList,Values),
            RBIT(tCacheStateBits,State,0) ] );
        ARRAY ( tCacheContents , CacheLines , tCacheLine );

        -- Definition of the memory array structure
        ARRAY ( tMemoryArray , MemorySize , int );

        -- Definition of the message structures
        ENUM ( tBusMessage , [BUS_REQUEST,BUS_GRANT,BUS_RELEASE] );
        STRUCT ( tBusReqMessage ,
            [RENUM(tBusMessage,MessageType,0),
```
RINT(Sender,0,HIDE));
STRUCT { t_Interrupt , [RINT(InterruptValue,0)] };
ARRAY ( t_BlockTransferUp , OutputUpBusWidth , int );
STRUCT ( t_BlockPacketUp , [RINT(Size,0),
    RARRAY(t_BlockTransferUp,DataBlock),
    RINT(Sender,0),RINT(MessageID,0),RINT(Source,0)] );
ARRAY ( t_BlockTransferDown , OutputDownBusWidth , int );
STRUCT ( t_BlockPacketDown , [RINT(Size,0),
    RARRAY(t_BlockTransferDown,DataBlock),
    RINT(Sender,0),RINT(MessageID,0),RINT(Source,0)] );
ARRAY ( t_MemoryBlock , OutputDownBusWidth , mt );
STRUCT ( t_MemoryPacketBlock , [RINT(Size,0),
    RARRAY(t_MemoryBlock,DataBlock)] );
STRUCT ( t_DataPacket , [RINT(Data,0), RINT(MessageID,0)] );
STRUCT ( t_RetryPacket , [RINT(Sender,0), RINT(Destination,0),
    RINT(MessageID,0)] );
STRUCT ( GenericPacket , [RINT(Address,0), RSTRING(Action,""),
    RINT(Size,0), RARRAY(t_BlockTransferUp,Data),
    RINT(Allocation,1), RINT(Sender,0), RINT(Destination,0),
    RINT(MessageID,0), RINT(Source,0)] );
STRUCT ( t_Protocol_Packet , [RINT(Address,0),
    RSTRING(Protocol_String,""),
    RSTRUCT(t_MemoryPacketBlock,Data,HIDE),
    RINT(Source,0),RINT(Sender,0),RINT(Destination,1),
    RINT(MessageID,0,HIDE)] );
STRUCT ( t_MemoryPacket , [RINT(Address,0),
    RSTRUCT(t_MemoryPacketBlock,Data),RINT(AllOcatiOfl,1,HIDE),
    RINT(Sender,1,HIDE), RINT(Destination,1,HIDE),
    RINT(MessageID,0,HIDE)] );

--Definition of the link parameter structures
LINK ( t_MemoryLink ,
    [REQ_READ,RSTRUCT(t_MemoryPacket,MemoryPacket)),
    (REQ_WRITE,RSTRUCT(t_MemoryPacket,MemoryPacket)),
    (COPYBACK_REQ,RSTRUCT(t_MemoryPacket,MemoryPacket)),
    (RESULT,RSTRUCT(t_BlockPacketDown,DataPacket)),
    (BUS_REQ,RSTRUCT(t_BusReq_Message, BusMessage)),
    (BUS_REL, RENUM(t_Bus Message, BusMessage, 0)),
    (RETRY_PACKET,RSTRUCT(t_RetryPacket,RetryPacket)),
    (PROTOCOL_PACKET,RSTRUCT(t_Protocol_Packet, Protocol_Packet))] )
LINK ( t_ResultLink ,
    [(RESULT,RSTRUCT(t_BlockPacketUp,DataPacket)),
    (WRITE_ACK,RSTRUCT(t_DataPacket,DataPacket)),
    (INSTRUCTION_ACK,RSTRUCT(t_DataPacket,DataPacket)),
    (RETRY_PACKET,RSTRUCT(t_RetryPacket,RetryPacket)),
    (BUS_GRT, RENUM(t_Bus Message, BusMessage, 0)),
    (PROTOCOL_PACKET,RSTRUCT(t_Protocol_Packet))] );
LINK ( t_InterruptLink ,
    [(INTERRUPT,RSTRUCT(t_Interrupt,Interrupt))]);
LINK ( t_GenericLink , [(GENERIC,RSTRUCT(GenericPacket,GP))] );
)
GLOBALS {
}
ENTITYLIB {

-- The execution-driven processor entity used to execute the lu
-- benchmark
ENTITY Processor (
  DESCRIPTION ("A Processor that executes lu")
  PARAMS (
    RINT(OutputDownBusWidth,1);
    RINT(Matrix_Dimension,128);
    RINT(Matrix_Block_Size,16);
    RINT(t_Synchronisation,Synchronisation,0);
    RINT(t_Memory_Consistency,Memory_Consistency,0);
    RINT(Instruction_Buffer_Size,1);
    RINT(LDI_delay,1);
    RINT(SUB_delay,1);
    RINT(ADD_delay,1);
    RINT(MULT_delay,3);
    RINT(ADD_delay,2);
    RINT(SUBD_delay,2);
    RINT(MULTD_delay,6);
    RINT(DIVD_delay,12);
    RINT(UBR_delay,1);
    RINT(CBR_delay,1);
  )
  PORTS (PORT (to_cache, t_MemoryLink, portdot, SOURCE);
          PORT(from_cache, t_ResultLink, portleft, DESTINATION);
  )
  ATTRIB ( )
);

-- The generic cache entity
ENTITY Cache (
  DESCRIPTION ("A generic cache component")
  PARAMS (RINT(OutputUpBusWidth,1);
           RINT(OutputDownBusWidth,4);
           RINT(BlockSize,4);
           RINT(CacheLines,1024);
           RARRAY(t_CacheContents,CacheContents);
           RENUM(t_AllocationPolicy,Allocation_Policy,0);
           RENUM(t_Write_Policy,Write_Policy,0);
           RENUM(t_Replacement_Policy,Replacement_Policy,1);
           RENUM(t_CacheState,CacheState,0);
           RENUM(t_Coherence_Protocol,Coherence_Protocol,5);
           RINT(Associativity,1);
           RINT(REQ_Tag,0);
           RINT(REQ_Index,0);
           RINT(REQ_BlockOffset,0);
           RINT(Hits,0);
           RINT(Misses,0);
           RINT(t_MemoryAction,CacheAction,3);
           RINT(Level,1);
           RINT(ReadDelay,1);
           RINT(WriteDelay,1);
  )
  PORTS (PORT (from_processor, t_MemoryLink, portright, DESTINATION);
          PORT (to_processor, t_ResultLink, portdot, SOURCE);
          PORT (to_memory, t_MemoryLink, portdot, SOURCE);
          PORT (from_memory, t_ResultLink, portleft, DESTINATION);
  )
)
-- A composite entity containing a processor and a cache used as
-- nodes to link to the bus
COMPENTITY Node {
  DESCENDANTS {
    CHILD(Processor, PROCESSOR, ATTRIB());
    CHILD(Cache, PRIMARY CACHE, ATTRIB());
    CLINK(Processor.PROCESSOR[to_cache] ->
      Cache.PRIMARY_CACHE[from_processor], 1);
    CLINK(Cache.PRIMARY_CACHE[to_process ->
      Processor.PROCESSOR[from_cache], 1];
  }
  DESCRIPTION("A Bus Node")
  PARAMS {
  }
  PORTS {
  }
  ATTRIB {
  }
};

-- The basic bus entity
ENTITY Bus {
  DESCRIPTION ("A Bus Component")
  PARAMS {
    RINT(OutputUpBusWidth, 4);
    RINT(OutputDownBusWidth, 4);
    RINT(BusCycle, 2);
    RINT(BlockSize, 4);
    RINT(BusMaster, -1);
    RENUM(t_Bus_State, Bus_State, 0);
    RENUM(t_Coherence_Protocol, Coherence_Protocol, 5);
    RENUM(t_Arbitration_Scheme, Arbitration_Scheme, 0);
  }
  PORTS {
  }
  ATTRIB {
  }
};

-- The memory entity, used to hold the data used by lu benchmark
ENTITY Memory {
  DESCRIPTION ("A memory to store the data")
  PARAMS {
    RINT(BlockSize, 4);
    RINT(OutputUpBusWidth, 4);
    RINT(MemorySize, 10000);
    RARRAY(t_MemoryArray, MemoryArray);
    RINT(MemoryReadDelay, 5);
    RINT(MemoryWriteDelay, 5);
    RENUM(t_MemoryAction, MemoryAction, 3);
    RENUM(t_Coherence_Protocol, Coherence_Protocol, 5);
  }
  PORTS {
    PORT(from_node, t_MemoryLink, portright, DESTINATION);
    PORT(to_node, t_ResultLink, portdot, SOURCE);
  }
-- One of the network interface entities, receives messages from the
-- internal network
ENTITY LowerLevelReceiver
    DESCRIPTION ("Deal with events from within cluster")
    PARAMS
        RINT(BlockSize,4);
        RINT(OutputDownBusWidth,4);
        RINT(OutputUpBusWidth,4);
        RENUM(t_Coherence_Protocol,Coherence_Protocol,5);
    PORTS
        PORT(from_bus,t_MemoryLink,portright,DESTINATION);
        PORT(to_sender,t_GenericLink,portdot,SOURCE);
        PORT(to_lower_sender,t_GenericLink,portdot,SOURCE);
    ATTRIB

-- One of the network interface entities, sends messages to the
-- internal network
ENTITY LowerLevelSender
    DESCRIPTION ("Entity to deal with events to bus")
    PARAMS
        RINT(BlockSize,4);
        RINT(OutputDownBusWidth,4);
        RINT(OutputUpBusWidth,4);
        RENUM(t_Coherence_Protocol,Coherence_Protocol_Lower,5);
    PORTS
        PORT(from_receiver,t_GenericLink,portleft,DESTINATION);
        PORT(to_bus,t_MemoryLink,portdot,SOURCE);
        PORT(from_lower_receiver,t_GenericLink,portdown,SOURCE);
    ATTRIB

-- One of the network interface entities, receives messages from the
-- external network
ENTITY UpperLevelReceiver
    DESCRIPTION ("Entity to deal with events from cluster")
    PARAMS
        RINT(BlockSize,4);
        RINT(OutputUpBusWidth,4);
        RINT(OutputDownBusWidth,4);
        RENUM(t_Coherence_Protocol,Coherence_Protocol_Upper,10);
    PORTS
        PORT(from_network,t_MemoryLink,portleft,DESTINATION);
        PORT(to_sender,t_GenericLink,portdot,SOURCE);
        PORT(to_upper_sender,t_GenericLink,portdot,SOURCE);
    ATTRIB
-- One of the network interface entities, sends messages to the
-- external network
ENTITY UpperLevelSender
  DESCRIPTION ("Entity to sender events to network")
  PARAMS
    RINT(BlockSize,4);
    RINT(OutputUpBusWidth,4);
    RINT(OutputDownBusWidth,4);
    RENUM(t_Coherence_Protocol,Coherence_Protocol,10);
  PORTS
    PORT(from_receiver,t_GenericLink,portright,DESTINATION);
    PORT(to_network,t_MemoryLink,portdot,SOURCE);
    PORT(from_upper_receiver,t_GenericLink,portup,SOURCE);
  ATTRIB
    );

-- The combined network interface entity, containing all four
-- interface entities
COMPENTITY NetworkInterface
  DESCENDANTS
    CHILD(LowerLevelReceiver,LOWERLEVELRECEIVER,ATTRIB());
    CHILD(LowerLevelSender,LOWERLEVELSENDER,ATTRIB());
    CHILD(UpperLevelReceiver,UPPERLEVELRECEIVER,ATTRIB());
    CHILD(UpperLevelSender,UPPERLEVELSENDER,ATTRIB());
    CLINK(LowerLevelReceiver.LOWERLEVELRECEIVER[to_sender] ->
      UpperLevelSender.UPPERLEVELSENDER[from_receiver],1);
    CLINK(UpperLevelReceiver.UPPERLEVELRECEIVER[to_sender] ->
      LowerLevelSender.LOWERLEVELSENDER[from_receiver],1);
    CLINK(LowerLevelReceiver.LOWERLEVELRECEIVER[to_lower_sender] ->LowerLevelSender.LOWERLEVELSENDER[from_lower_receiver],1);
    CLINK(UpperLevelReceiver.UPPERLEVELRECEIVER[to_upper_sender] ->UpperLevelSender.UPPERLEVELSENDER(from_upper_receiver),1);
  DESCRIPTION ("An interface between the two networks")
  PARAMS
  PORTS
  ATTRIB
  );

-- An interconnection network that has defined number of ports, but
-- can be programmed in to behave in any manner by providing a
-- different behavioural specification in the .hase file. Can be
-- used to construct multiprocessors but requires work to add or
-- remove nodes.
ENTITY InterconnectionNetwork
  DESCRIPTION ("An inflexible interconnection network")
  PARAMS
    RINT(OutputDownBusWidth,4);
RINT(OutputUpBusWidth, 4);
RINT(BlockSize, 4);
RINT(NetworkDelay, 10);
)
PORTS (  
    PORT(to_cluster0, t_MemoryLink, portdot, SOURCE);
    PORT(from_cluster0, t_MemoryLink, portright, DESTINATION);
    PORT(to_cluster1, t_MemoryLink, portdot, SOURCE);
    PORT(from_cluster1, t_MemoryLink, portright, DESTINATION);
    PORT(to_cluster2, t_MemoryLink, portdot, SOURCE);
    PORT(from_cluster2, t_MemoryLink, portright, DESTINATION);
    PORT(to_cluster3, t_MemoryLink, portdot, SOURCE);
    PORT(from_cluster3, t_MemoryLink, portright, DESTINATION);
  )
ATTRIB (  
    )
);

-- A parameterised bus-based shared-memory multiprocessor, with
-- processor-cache pairs as nodes.
BUSENTITY BusMultiprocessor (  
    NODE(Node)
    MEMORY(Memory)
    BUS(Bus)
    NETWORKINTERFACE(NetworkInterface)
    NUMBERNODES(8)
    DESCRIPTION("Bus-based multiprocessor")
    PARAMS (  
        )
    PORTS (  
        )
    ATTRIB (  
        )
    )

-- A parameterised multiple common memory multiprocessor, with
-- processor-cache pairs as nodes
MULTIPLEMEMORY MultipleMemoryMultiprocessor (  
    NODE(Node)
    MEMORY(Memory)
    NETWORK(CrossBar)
    NUMBERNODES(8)
    NUMBERMEMORIES(8)
    DESCRIPTION("A multiple memory multiprocessor")
    PARAMS (  
        )
    PORTS (  
        )
    ATTRIB (  
        )
    )

-- A parameterised network entity with bus-based multiprocessors as
-- nodes, creating a clustered distributed shared-memory
-- multiprocessor
NETWORKENTITY ClusteredMultiprocessor (  
    NODE(BusMultiprocessor)
    NETWORK(CrossBar)
    NUMBERNODES(8)
    DESCRIPTION("A clustered distributed shared-mem multiproc")
    )
PARAMS {
    RINT(OutputUpBusWidth, 4);
    RINT(OutputDownBusWidth, 4);
}

PORTS {
    
}

ATTRIB {
    
};

-- A clustered distributed shared-memory multiprocessor constructed
-- from the fixed interconnection network. This requires work to
-- change the number of nodes.
COMPENTITY FixedCluster {
    DESCENDANTS {
        CHILD(InterconnectionNetwork, NETWORK, ATTRIB());
        CHILD(BusMultiprocessor, Cluster0, ATTRIB());
        CHILD(BusMultiprocessor, Cluster1, ATTRIB());
        CHILD(BusMultiprocessor, Cluster2, ATTRIB());
        CHILD(BusMultiprocessor, Cluster3, ATTRIB());
        CLINK(InterconnectionNetwork.NETWORK[to_cluster0] ->
                BusMultiprocessor.Cluster0[from_network], 1);
        CLINK(BusMultiprocessor.Cluster0[to_network] ->
                InterconnectionNetwork.NETWORK[from_cluster0], 1);
        CLINK(InterconnectionNetwork.NETWORK[to_cluster1] ->
                BusMultiprocessor.Cluster1[from_network], 1);
        CLINK(BusMultiprocessor.Cluster1[to_network] ->
                InterconnectionNetwork.NETWORK[from_cluster1], 1);
        CLINK(InterconnectionNetwork.NETWORK[to_cluster2] ->
                BusMultiprocessor.Cluster2[from_network], 1);
        CLINK(BusMultiprocessor.Cluster2[to_network] ->
                InterconnectionNetwork.NETWORK[from_cluster2], 1);
        CLINK(InterconnectionNetwork.NETWORK[to_cluster3] ->
                BusMultiprocessor.Cluster3[from_network], 1);
        CLINK(BusMultiprocessor.Cluster3[to_network] ->
                InterconnectionNetwork.NETWORK[from_cluster3], 1);
    }
    DESCRIPTION("Fixed size clustered multiprocessor")
}

PARAMS {
    RINT(NetworkDelay, 10);
}

PORTS {
    
}

ATTRIB {
    
};

STRUCTURE {
-- The structure section contains the necessary code to create the
-- different architectures described in the thesis. To include the
-- appropriate architecture uncomment the necessary pieces of EDL
-- A simple single processor system
-- AENTITY Node NODE (DESCRIPTION("Uniprocessor node") ATTRIB());
-- AENTITY Memory MEMORY (DESCRIPTION("Simple Memory") ATTRIB());
-- CLINK(Node.NODE[to_memory].Memory.MEMORY[from_node], 1);
-- CLINK(Memory.MEMORY[to_node].Node.NODE[from_memory], 1);
-- A bus-based shared-memory multiprocessor

-- AENTITY BusMultiprocessor BUSMULTIPROCESSOR(DESCRIPTION("A bus-
-- based shared-memory multiprocessor") ATTRIB ());

-- A multiple common memory multiprocessor
-- AENTITY MultipleMemoryMultiprocessor MMMPROCESSOR(DESCRIPTION(" 
-- A multiple common memory multiprocessor") ATTRIB ());

-- A cluster distributed shared-memory multiprocessor
-- AENTITY ClusteredMultiprocessor CLUSTEREDMULTIPROCESSOR
-- (DESCRIPTION("A clustered distributed shared-memory 
-- multiprocessor") ATTRIB ());

-- A fixed size clustered distributed shared-memory multiprocessor
-- AENTITY FixedCluster FIXEDCLUSTER (DESCRIPTION("A fixed size 
-- clustered distributed shared-memory multiprocessor")
-- ATTRIB ());
Appendix C

This appendix contains the code used to extend the C code of the \textit{lu} program to include the necessary delays to synchronise it with the simulation. For each function the original code is presented first, followed by an implementation in assembly language. Finally, the extended code with the appropriate delay counters is shown.

C.1 The \textit{lu0} Function

C.1.1 C Code for \textit{lu0}

C Code for \textit{lu0}

\begin{verbatim}
for (k=0;k<n;k++) {
    for (j=k+1;j<n;j++) {
        a[k+j*stride] /= a[k+k*stride];
        alpha -= a[k+j*stride];
        daxpy(&a[k+j+1*stride],&a[k+1+k*stride],fl_k_1,alpha);
    }
}
\end{verbatim}

C.1.2 Assembly code for \textit{lu0}

\begin{verbatim}
.LI k 0
.StartK:
    SUB tmp n k
    BLEZ tmp EndK
    ADD j k 1
.StartJ:
    SUB tmp n j
    BLEZ tmp EndJ
    MULT a il k stride
    ADD a_il a il k
    ADD a_addr a_base a il
    LD al a_addr
    MULT a_i2 j stride
    ADD a_i2 a_i2 k
    ADD a_addr base_a a_i2
    LD a2 a_addr
    DIVD a2 a2 a1
    SD a2 a_addr
    SUBD alpha 0 a2
    ADD a_i2_pl a_i2 1
    ADD a_i2_pl a_i2_pl base_a
    ADD a_il_pl a il 1
    ADD a_il_pl a il_pl base_a
\end{verbatim}

# Initialise k to 0
# Calculate n-k
# If <0 finished loop
# Initialise j to k+1
# Calculate n-j
# If <0 finished loop
# Calculate k*stride
# Calculate k+k*stride
# Find address of element
# Load element into al
# Calculate j*stride
# Calculate k+j*stride
# Find address of element
# Load element into a2
# Calculate a2/a1
# Write a2 into a_addr
# Calculate -a[k+j]*stride
# Calculate k+1+j*stride
# Find address of element
# Calculate k+1+k*stride
# Find address of element
C.1.3 Extended C Code for luO

```c
hold_counter+=LDI_delay;
for (k=0;k<n;k++) {
    hold_counter+=SUB_delay+CBR_delay;
    hold_counter+=ADD_delay;
    for (j=k+1;j<n;j++) {
        hold_counter+=SUB_delay+CBR_delay;
        hold_counter+=2*MULT_delay+4*ADD_delay;
        a_k_j=Read(a[k+j*stride]);
        a_k_k=Read(a[k+k*stride]);
        hold_counter+=DIVD_delay
        a_k_j/=a_k_k;
        Write(a[k+j*stride],a_k_j);
        hold_counter+=SUBD_delay;
        alpha=-a_k_j;
        hold_counter+=4*ADD_delay+2*SUB_delay;
        daxpy(&a[k+1+j*stride],&a[k+1+k*stride],n-k-1,alpha);
        hold_counter+=ADD_delay+UBR_delay;
    }
    hold_counter+=ADD_delay+UBR_delay;
}
```

C.2 The bdiv Function

C.2.1 C Code for bdiv

```c
for (k=0;k<dimk;k++) {
    for (j=k+1;j<dimk;j++) {
        alpha=diag[k+j*stridediag];
        daxpy(&a[j*stride_a],&a[k*stride_a],dimi,alpha);
    }
}
```

C.2.2 Assembly Code for bdiv

```assembly
    LI k 0     # Initialise k to 0
  StartK:
    SUB tmp dimk k  # Calculate dimk-k
    BLEZ tmp EndK  # If <0 finished loop
    ADD j k l     # Initialise j to k+1
```
C.2.3 Extended C Code for bdiv

hold_counter+=LDI_delay
for (k=0;k<dimk;k++) {
    hold_counter+=SUB_delay+CBR_delay;
    hold_counter+=ADD_delay;
    for (j=k+1;j<dimk;j++) {
        hold_counter+=SUB_delay+CBR_delay;
        hold_counter+=MULT_delay+2*ADD_delay;
        alpha=-(Read(diag[k+j*stride_diag]));
        hold_counter+=SUBD_delay;
        hold_counter+=2*MULT_delay+2*ADD_delay;
        daxpy(&a[j*stride_a],&a[k*stride_a],dimi,alpha);
        hold_counter+=ADD_delay+UBR_delay;
    }
    hold_counter+=ADD_delay+UBR_delay;
}

C.3 The bmodd Function

C.3.1 C Code for bmodd

for (k=0;k<dimi;k++) {
    for (j=0;j<dimj;j++) {
        c[k+j*stride_c]=a[k+k*stride_a];
        alpha=-c[k+j*stride_c];
        daxpy(&c[k+l+j*stride_c],&a[k+l+k*stride_a],dimi-k-1,alpha);
    }
}
C.3.2 Assembly Code for bmodd

```
LI k 0    # Initialise k to 0
StartK:  SUB tmp dimi k    # Calculate dimi-k
   BLEZ tmp EndK    # If <0 finished loop
   LI j 0    # Initialise j to 0
StartJ:  SUB tmp dimj j    # Calculate dimj-j
   BLEZ tmp EndJ    # If <0 finished loop
   MULT a_i k stride_a    # Calculate k*stride_a
   ADD a_i_a_i k    # Calculate k+k*stride_a
   ADD a_addr base_a a_i    # Find address of element
   LD a a_addr    # Load element into a
   MULT c_i j stride_c    # Calculate k+j*stride_c
   ADD c_i_c_i k    # Calculate k+j+k*stride_c
   ADD c_addr c_base c_i    # Find address of element
   LD c c_addr    # Load element into c
   DIVD c c a    # Calculate c/a
   SD c c_addr    # Write c to c_addr
   SUBD alpha 0 c    # Calculate -c[k+j*stride_c]
   ADD c_i_c_i l    # Calculate k+l+j*stride_c
   ADD c_i base_c c_i    # Find address of element
   ADD a_i_a_i l    # Calculate k+l+k*stride_a
   ADD a_i base_a a_i    # Find address of element
   SUB tmp dimi k    # Calculate dimi-k
   SUB tmp tmp 1    # Calculate dimi-k-1
   CALL_DAXPY    # Call daxpy function
   ADD j j 1    # Increment j
   BR StartJ    # Return to start of loop
EndJ:    ADD k k 1    # Increment k
         BR StartK    # Return to start of loop
EndK:    Finished    # Completed bmodd function
```

C.3.3 Extended C Code for bmodd

```
hold_counter+=LDI_delay;
for(k=0;k<dimi;k++) {
    hold_counter+=SUB_delay+CBR_delay;
    hold_counter+=LDI_delay;
    for (j=0;j<dimj;j++) {
        hold_counter+=SUB_delay+CBR_delay;
        hold_counter+=2*MULT_delay+4*ADD_delay;
        c_k_j=Read(c[k+j*stride_c]);
        a_k_k=Read(a[k+k*stride_a]);
        hold_counter+=DIVD_delay
        c_k_j/=a_k_k;
        Write(c[k+j*stride_c],c_k_j);
        alpha=-c_k_j;
        hold_counter+=SUBD_delay;
        hold_counter+=4*ADD_delay+2*SUB_delay;
        daxpy(sc[k+1+j*stride_c],a[k+1+k*stride_a],dimi-k-1,alpha);
    }
    hold_counter+=ADD_delay+UBR_delay;
}
hold_counter+=ADD_delay+UBR_delay;
```
C.4 The bmod Function

C.4.1 C Code for bmod

```c
for (k=0; k<dimk; k++) {
    for (j=0; j<dimj; j++) {
        alpha = -b[k+j*stride];
        daxpy(&c[j*stride], &a[k*stride], dimi, alpha);
    }
}
```

C.4.2 Assembly Code for bmod

```assembly
LI k 0
StartK: SUB tmp dimk k
BLEZ tmp EndK
LI j 0
StartJ: SUB tmp dimj j
BLEZ tmp EndJ
MULT j s_j stridé
ADD b il j s k
ADD b_addr b_base b il
LD b b_addr
SUBD alpha 0 b
ADD c il base_c j s
MULT a il k stride
ADD a il base_a a il
CALL DAXPY
ADD j j l
BR StartJ
EndJ: ADD k k 1
BR StartK
EndK: Finished
```

C.4.3 Extended C Code for bmod

```c
hold_counter+=LDI_delay;
for (k=0; k<dimk; k++) {
    hold_counter+=SUB_delay+CBR_delay;
    hold_counter+=LDI_delay;
    for (j=0; j<dimj, j++) {
        hold_counter+=SUB_delay+CBR_delay;
        hold_counter+=MULT_delay+2*ADD_delay;
        alpha=-(Read(b[k+j*stride]));
        hold_counter+=SUBD_delay;
        hold_counter+=MULT_delay+2*ADD_delay;
        daxpy(&c[j*stride], &a[k*stride], dimi, alpha);
        hold_counter+=ADD_delay+UBR_delay;
    }
}
```
C.5 The daxpy Function

C.5.1 C Code for daxpy

```c
for (i=0;i<n;i++) {
    a[i]+=alpha*b[i];
}
```

C.5.2 Assembly Code for daxpy

```
LI i 0
Start1: SUB tmp n i
        BLEZ tmp End1
        ADD b_addr i base_b
        LD b b_addr
        ADD a_addr i base_a
        LD a a_addr
        MULTD tmp alpha b
        ADDD a a tmp
        SD a a_addr
        ADD i i 1
        BR Start1
End1:   Finished
```

# Initialise i to 0
# Calculate n-i
# If <0 finished loop
# Find address of element
# Load element into b
# Find address of element
# Load element into a
# Calculate alpha*b
# Calculate a+alpha*b
# Write a to a_addr
# Increment i
# Return to start of loop
# Completed daxpy function

C.5.3 Extended C Code for daxpy

```c
hold_counter+=LDI_delay;
for (i=0;i<n;i++) {
    hold_counter+=SUB_delay+CBR_delay;
    hold_counter+=2*ADD_delay;
    b_i=Read(b[i]);
    hold_counter+=MULTD_delay+ADDD_delay;
    Write(a[i],alpha*b_i);
    hold_counter+=ADD_delay+UBR_delay;
}
```

C.6 The blockowner Function

C.6.1 C Code for blockowner

```c
return (i*num_cols)+(j*num_rows)*num_cols;
```
C.6.2 Assembly Code for blockowner

DIVD tmp1 i nuincols  # Calculate i/num_cols
MULTD tmp1 tmp1 num_cols  # Calculate tmp1*num_cols
SUBD tmp1 i tmp  # Calculate i%num_cols
DUVD tmp2 j num_rows  # Calculate j/num_rows
MULTD tmp2 tmp2 num_rows  # Calculate tmp2*num_rows
SUBD tmp2 j tmp2  # Calculate j%num_rows
MULTD tmp2 tmp2 num_cols  # Calculate tmp2*num_cols
ADDD tmp1 tmp1 tmp2  # Calculate result

C.6.3 Extended C Code for blockowner

hold_counter+=2*DIVD delay+3*MULTD delay+2*SUBD delay+ADDD delay;
return (i%num_cols)+(j%num_rows)*num_cols;

C.7 The lu Function

C.7.1 C Code for lu

for (k=0;k<n;k+=bs,K++) {
    kl=k+bs;
    if (k1>n)
        kl=n;
    diagowner=blockowner(K,K);
    if (diagowner==MyNum) {
        A=&(a[k+k*n]);
        lu0(A,kl-k,strI);
    }
    BARRIER();
    D=&(a[k+k*n]);
    for (i=kl,I=K+l;i<n;i+=bs,I++) {
        if (blockowner(I,K)==MyNum) {
            il=i+bs;
            if (il>n)
                il=n;
            A=&(a[i+k*n]);
            bdiv(A,D,strI,n,il-i,kl-k);
        }
    }
    for (j=kl,J=K+l;j<n;j+=bs,J++) {
        if (blockowner(K,J)==MyNum) {
            jl=j+bs;
            if (jl>n)
                jl=n;
            A=&(a[k+j*n]);
            bmod(D,A,kl-k,jl-j,n,strI);
        }
    }
}
C.7.2 Assembly Code for lu

```
LI k 0
LI K 0

StartLK: SUB tmp n k
BLEZ tmp EndLK:
ADD kl k bs
SUB tmp kl n
BLEZ tmp EndIfKl
ADD kl n 0

EndIfKl: diagowner=blockowner
SUB tmp diagowner MyNum
BNEZ tmp EndIfD
MULT a_il k n
ADD a_il a_il k
ADD a_addr base_a a_il
SUB tmpk kl k
CALL lu0

EndIFD: BARRIER
MULT d_il k n
ADD d_il d_il k
ADD d_addr base_a d_il
ADD i^k1 0
ADD I K 1

StartLII: SUB tmp n i
BLEZ tmp EndLII
SUB tmp MyNum blockowner
BNEZ tmp EndIfMNl
ADD il i bs
SUB tmp il n
BLEZ tmp EndIfIl
ADD il n 0

EndIfIl: MULT a_il k n
ADD a_il a_il i
ADD a_addr base_a a_il
SUB tmpi il i
SUB tmpk kl k
```
CALL bdiv
# Call bdiv function
EndIfMN1: ADD i i bs
# Calculate i+bs
ADD I I 1
# Increment I
BR StartLI1
# Return to start of loop
EndLI1: ADD j kl 0
# Initialise j to kl
ADD J K 1
# Initialise J to K+1
StartLI1: SUB tmp n j
# Calculate n-j
BLEZ tmp EndLI1
# If <0 finished loop
SUB tmp MyNum blockowner
# Calculate MyNum-blockowner
BNEZ tmp EndIfMN2
# If !=0 skip if
ADD jl j bs
# Calculate j+bs
SUB tmp jl n
# Calculate jl-n
BLEZ tmp EndIfJ1
ADD j l n 0
# Set jl to n
EndIfJ1: MULT a il j n
# Calculate j*n
ADD a il a il k
# Calculate k+j*n
ADD a_addr base_a a il
# Find address of element
SUB tmp i il i
# Calculate il-i
SUB tmpj jl j
# Calculate jl-j
CALL bmodd
# Call bmodd function
EndIfMN2: ADD j j bs
# Calculate j+bs
ADD J J 1
# Increment J
BR StartLJ1
# Return to start of loop
EndLJ1: BARRIER
# Barrier Synchronise
ADD i kl 0
# Initialise i to kl
ADD I K 1
# Initialise I to K+1
StartLJ2: SUB tmp n i
# Calculate n-i
BLEZ tmp EndLJ2
# If <0 finished loop
ADD il i bs
# Calculate i+bs
SUB tmp il n
# Calculate il-n
BLEZ tmp EndIfJ2
# If <0 skip if
ADD il n 0
# Set il to n
EndIfJ2: MULT a il k n
# Calculate k*n
ADD a il a il i
# Calculate i+k*n
ADD a_addr base_a a il
# Find address of element
ADD j kl 0
# Initialise j to kl
ADD J K 1
# Initialise J to K+1
StartLJ2: SUB tmp n j
# Calculate n-j
BLEZ tmp EndLJ2
# If <0 finished loop
ADD jl j bs
# Calculate j+bs
SUB tmp jl n
# Calculate jl-n
BLEZ tmp EndIfJ2
# If <0 skip if
ADD jl n 0
# Set jl to n
EndIfJ2: SUB tmp MyNum blockowner
# Calculate MyNum-blockowner
BNEZ tmp EndIfMN3
# If !=0 skip if
MULT a il j n
# Calculate j*n
ADD b il a il k
# Calculate k+j*n
ADD b_addr base_a b il
# Find address of element
ADD c il a il j
# Calculate j+j*n
ADD c_addr base_a c il
# Find address of element
SUB tmp il il i
# Calculate il-i
SUB tmpj jl j
# Calculate jl-j
SUB tmpk kl k
# Calculate k1-k
CALL bmod
# Call bmod function
EndIfMN3: ADD j j bs
# Calculate j+bs
ADD J J 1
# Increment J
BR StartLJ2
# Return to start of loop
EndLJ2: ADD i i bs
# Calculate i+bs
ADD I I 1
# Increment I
C.7.3 Extended C Code for lu

```c
hold_counter+=2*LDI_delay;
for (k=0,K=0;k<n;k+=bs,K++) {
    hold_counter+=SUB_delay+CBR_delay;
    hold_counter+=ADD_delay;
    kl=k+bs;
    hold_counter+=SUB_delay+CBR_delay;
    if (kl>n) {
        hold_counter+=ADD_delay;
        kl=n;
    }
    diagowner=blockowner(K,K);
    hold_counter+=SUB_delay+CBR_delay);
    if (diagowner==MyNum) {
        hold_counter+=MULT_delay+2*ADD_delay;
        A=&(a[k+k*n]);
        hold_counter+=SUB_delay;
        lu0(A,kl-k,strl);
        BARRIER()
    }
    hold_counter+=MULT_delay+2*ADD_delay;
    D=&(a[k+k*n]);
    hold_counter+=2*ADD_delay;
    for (i=kl,I=K+1;i<n;i+=bs,I++) {
        hold_counter+=SUB_delay+CBR_delay;
        hold_counter+=SUB_delay+CBR_delay;
        if (blockowner(I,K)==MyNum) {
            hold_counter+=ADD_delay;
            il+=i+bs;
            hold_counter+=SUB_delay+CBR_delay;
            if (il>n) {
                hold_counter+=ADD_delay;
                il=n;
            }
            hold_counter+=MULT_delay+2*ADD_delay;
            A=&(a[i+k*n]);
            hold_counter+=2*SUB_delay;
            bdiv(A,D,strl,n,il-i,kl-k);
        }
        hold_counter+=2*ADD_delay+UBR_delay;
    }
    hold_counter+=2*ADD_delay;
    for (j=kl,J=K+1;j<n;j+=bs,J++) {
        hold_counter+=SUB_delay+CBR_delay;
        hold_counter+=SUB_delay+CBR_delay;
        if (blockowner(K,J)==MyNum) {
            hold_counter+=ADD_delay;
            jl=j+bs;
            hold_counter+=SUB_delay+CBR_delay;
        }
    }
}
```

272
if (j1>n) {
    hold_counter+=ADD_delay;
    j1=n;
}
hold_counter+=MULT_delay+2*ADD_delay;
A=&(a[k+j*n]);
hold_counter+=2*SUB_delay;
bmodd(D,A,kl-k,j1-j,n,strI);
hold_counter+=2*ADD_delay+UBR_delay;
}
BARRIER();
hold_counter+=2*ADD_delay;
for (i=kl,I=K+1;i<n;i+=bs,I++) {
    hold_counter+=SUB_delay+CBR_delay;
    hold_counter+=ADD_delay;
    i1=i+bs;
    hold_counter+=SUB_delay+CBR_delay;
    if (i1>n) {
        hold_counter+=ADD_delay;
        i1=n;
    }
    hold_counter+=MULT_delay+2*ADD_delay;
    A=&(a[i+k*n]);
    hold_counter+=2*ADD_delay;
    for (j=kl,J=K+1;j<n;j+=bs,J++) {
        hold_counter+=SUB_delay+CBR_delay;
        hold_counter+=ADD_delay;
        j1=j+bs;
        hold_counter+=SUB_delay+CBR_delay;
        if (j1>n) {
            hold_counter+=ADD_delay;
            j1=n;
        }
    }
    hold_counter+=SUB_delay+CBR_delay;
    if (blockowner(I,J)==MyNum) {
        hold_counter+=MULT_delay+2*ADD_delay;
        B=&(a[k+j*n]);
        hold_counter+=2*ADD_delay;
        C=(a[i+j*n]);
        hold_counter+=3*SUB_delay;
        bmod(A,B,C,i1-i,j1-j,kl-k,n);
    }
    hold_counter+=2*ADD_delay+UBR_delay;
}
hold_counter+=2*ADD_delay+UBR_delay;
}
hold_counter+=2*ADD_delay+UBR_delay;
}
C.8  The SlaveStart Function

C.8.1  C Code for SlaveStart

LOCK()
MyNum=GlobalID;
GlobalID++;
UNLOCK();
OneSolve();

C.8.2  Assembly Code for SlaveStart

LOCK
LD MyNum GlobalID
ADD GlobalID GlobalID 1
UNLOCK
CALL_ONESOLVE

# Request lock
# Load MyNum from GlobalID
# Increment GlobalID
# Release lock
# Call OneSolve function

C.8.3  Extended C Code for SlaveStart

LOCK();
MyNum=Read(GlobalID);
hold_counter+=ADD_delay;
Write(GlobalID,MyNum+1);
UNLOCK();
OneSolve();
Appendix D

This appendix contains a list of the parameter values that affect the operation of the simulation for each of the experiments performed. A table of results is also included for each experiment.

D.1 Cache Associativity and Block Replacement Policy Experiment

D.1.1 Parameter Values

Processor
OutputDownBusWidth – 1
Matrix_Dimension – 128, Matrix_Block_Size – 16
Synchronisation – CentralBarrier
Memory_Consistency – Sequential_WeakOrder, Instruction_Buffer_Size – 1
LDI_delay – 1, SUB_delay – 1, ADD_delay – 1, MULT_delay – 3,
ADDD_delay – 2, SUBD_delay – 2, MULTD_delay – 6, DIVD_delay – 12,
UBR_delay – 1, CBR_delay – 1

Cache
OutputUpBusWidth – 1, OutputDownBusWidth – 4
BlockSize – 4, CacheLines – 512
Allocation_Policy – WRITE_ALLOC, Write_Policy – COPY_BACK
Replacement_Policy – Random, LRU and Round_Robin
Associativity – 1, 2, 4, 8 and 0
Coherence_Protocol – NoProtocol
Level – 1
ReadDelay – 1, WriteDelay – 1
Memory
OutputUpBusWidth – 4
BlockSize - 4
MemorySize – 40000
MemoryReadDelay – 5, MemoryWriteDelay – 5
Coherence_Protocol - NoProtocol

D.1.2 Table of Results

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Replacement Policy</th>
<th>Hit Rate (%)</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Mapped</td>
<td>Random</td>
<td>85.73</td>
<td>20661712</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>85.73</td>
<td>20661712</td>
</tr>
<tr>
<td></td>
<td>Round Robin</td>
<td>85.73</td>
<td>20661712</td>
</tr>
<tr>
<td>2-Way</td>
<td>Random</td>
<td>89.18</td>
<td>19466676</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>81.43</td>
<td>22006856</td>
</tr>
<tr>
<td></td>
<td>Round Robin</td>
<td>89.09</td>
<td>19564315</td>
</tr>
<tr>
<td>4-Way</td>
<td>Random</td>
<td>89.88</td>
<td>19246059</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>89.89</td>
<td>19422218</td>
</tr>
<tr>
<td></td>
<td>Round Robin</td>
<td>87.53</td>
<td>19975590</td>
</tr>
<tr>
<td>8-Way</td>
<td>Random</td>
<td>90.19</td>
<td>19138708</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>90.18</td>
<td>19346290</td>
</tr>
<tr>
<td></td>
<td>Round Robin</td>
<td>83.94</td>
<td>20912454</td>
</tr>
<tr>
<td>Fully Associative</td>
<td>Random</td>
<td>98.00</td>
<td>15758419</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
<td>96.88</td>
<td>16241417</td>
</tr>
<tr>
<td></td>
<td>Round Robin</td>
<td>98.36</td>
<td>15599394</td>
</tr>
</tbody>
</table>
D.2 Write Policy and Allocation Policy Experiment

D.2.1 Parameter Values

Processor
OutputDownBusWidth – 1
Matrix_Dimension – 128, Matrix_Block_Size – 16
Synchronisation – CentralBarrier
Memory_Consistency – Sequential_WeakOrder, Instruction_Buffer_Size – 1
LDI_delay – 1, SUB_delay – 1, ADD_delay – 1, MULD_delay – 3,
ADDD_delay – 2, SUBD_delay – 2, MULTD_delay – 6, DIVD_delay – 12,
UBR_delay – 1, CBR_delay – 1

Cache
OutputUpBusWidth – 1, OutputDownBusWidth – 4
BlockSize – 4, CacheLines – 32, 128, 512, 2048, 8192 and 32768
Allocation_Policy – WRITE_ALLOC and NO_WRITE_ALLOC
Write_Policy – COPY_BACK and WRITE_THROUGH
Replacement_Policy – LRU, Associativity – 1
Coherence_Protocol – NoProtocol
Level – 1
ReadDelay – 1, WriteDelay – 1

Memory
OutputUpBusWidth – 4
BlockSize - 4
MemorySize – 40000
MemoryReadDelay – 5, MemoryWriteDelay – 5
Coherence_Protocol - NoProtocol
## D.2.2 Table of Results

<table>
<thead>
<tr>
<th>Write Policy/ Allocation Policy</th>
<th>Cache Size (Lines)</th>
<th>Cache Size (KB)</th>
<th>Hit Rate (%)</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Copy-Back/ Write Allocate</strong></td>
<td>32</td>
<td>0.5</td>
<td>58.39</td>
<td>29369740</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>2</td>
<td>70.53</td>
<td>25334484</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>85.73</td>
<td>20661712</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>32</td>
<td>96.92</td>
<td>16098057</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>128</td>
<td>99.81</td>
<td>15054354</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.81</td>
<td>15052469</td>
</tr>
<tr>
<td><strong>Write-Through/ No Write Allocate</strong></td>
<td>32</td>
<td>0.5</td>
<td>33.70</td>
<td>26603995</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>2</td>
<td>61.73</td>
<td>23527171</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>83.17</td>
<td>21109159</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>32</td>
<td>95.94</td>
<td>18578805</td>
</tr>
<tr>
<td></td>
<td>8192</td>
<td>128</td>
<td>99.05</td>
<td>17981137</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.05</td>
<td>17980751</td>
</tr>
</tbody>
</table>
D.3 Multiple Levels of Cache Experiment

D.3.1 Parameter Values

**Processor**
- OutputDownBusWidth = 1
- Matrix_Dimension = 128, Matrix_Block_Size = 16
- Synchronisation = CentralBarrier
- Memory_Consistency = Sequential_WeakOrder, Instruction_Buffer_Size = 1
- LDI_delay = 1, SUB_delay = 1, ADD_delay = 1, MULT_delay = 3,
  ADDD_delay = 2, SUBD_delay = 2, MULTD_delay = 6, DIVD_delay = 12,
  UBR_delay = 1, CBR_delay = 1

**Level 1 Cache**
- OutputUpBusWidth = 1, OutputDownBusWidth = 4
- BlockSize = 4, CacheLines = 32
- Allocation_Policy = WRITE_ALLOC, Write_Policy = COPY_BACK
- Replacement_Policy = LRU, Associativity = 1
- Coherence_Protocol = NoProtocol
- Level = 1
- ReadDelay = 1, WriteDelay = 1

**Level 2 Cache**
- OutputUpBusWidth = 4, OutputDownBusWidth = 4
- BlockSize = 4, CacheLines = 64, 256, 1024, 4096 and 16384
- Allocation_Policy = WRITE_ALLOC, Write_Policy = COPY_BACK
- Replacement_Policy = LRU, Associativity = 1
- Coherence_Protocol = NoProtocol
- Level = 2
- ReadDelay = 2, 4, 6 and 8, WriteDelay = 2, 4, 6 and 8

**Memory**
- OutputUpBusWidth = 4
- BlockSize = 4
MemorySize – 40000
MemoryReadDelay – 10, MemoryWriteDelay – 10
Coherence_Protocol - NoProtocol

D.3.2 Table of Results

<table>
<thead>
<tr>
<th>Access Time</th>
<th>Size (Lines)</th>
<th>Size (KB)</th>
<th>Hit Rate (%)</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>64</td>
<td>1</td>
<td>13.58</td>
<td>33673943</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>4</td>
<td>40.15</td>
<td>28957277</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>16</td>
<td>84.72</td>
<td>21043921</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>94.57</td>
<td>19270633</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>256</td>
<td>99.35</td>
<td>18416653</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>1</td>
<td>13.58</td>
<td>37832505</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>4</td>
<td>40.15</td>
<td>32954312</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>16</td>
<td>84.72</td>
<td>23671345</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>94.57</td>
<td>21586736</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>256</td>
<td>99.35</td>
<td>20560445</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>1</td>
<td>13.58</td>
<td>42937442</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>4</td>
<td>40.15</td>
<td>37464359</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>16</td>
<td>84.72</td>
<td>26818284</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>94.57</td>
<td>24478452</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>256</td>
<td>99.35</td>
<td>23296652</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>1</td>
<td>13.58</td>
<td>48299848</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>4</td>
<td>40.15</td>
<td>42251825</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>16</td>
<td>84.72</td>
<td>30268066</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>94.57</td>
<td>27687644</td>
</tr>
<tr>
<td></td>
<td>16384</td>
<td>256</td>
<td>99.35</td>
<td>26352928</td>
</tr>
<tr>
<td>No L2 Cache</td>
<td>32</td>
<td>0.5</td>
<td>58.39</td>
<td>41931054</td>
</tr>
</tbody>
</table>

280
D.4 Cache Coherence Protocol Experiment

D.4.1 Parameter Values

Processor
OutputDownBusWidth = 1
Matrix_Dimension = 128, Matrix_Block_Size = 16
Synchronisation = CentralBarrier
Memory_Consistency = Sequential_WeakOrder, Instruction_Buffer_Size = 1
LDI_delay = 1, SUB_delay = 1, ADD_delay = 1, MULT_delay = 3,
ADDD_delay = 2, SUBD_delay = 2, MULTD_delay = 6, DIVD_delay = 12,
UBR_delay = 1, CBR_delay = 1

Cache
OutputUpBusWidth = 1, OutputDownBusWidth = 4
BlockSize = 4, CacheLines = 64, 512, 4096 and 32768
Allocation_Policy = WRITE_ALLOC (NO_WRITE_ALLOC for Classical protocol)
Write_Policy = COPY_BACK (WRITE_THROUGH for Classical protocol)
Replacement_Policy = LRU, Associativity = 1
Coherence_Protocol = Classical, WriteOnce, Mesi, MOESI, Synapse, Berkeley, Illinois, Firefly and Dragon
Level = 1, ReadDelay = 1, WriteDelay = 1

Memory
OutputUpBusWidth = 4
BlockSize = 4, MemorySize = 40000
MemoryReadDelay = 5, MemoryWriteDelay = 5
Coherence_Protocol = Classical, WriteOnce, Mesi, MOESI, Synapse, Berkeley, Illinois, Firefly and Dragon

Bus
OutputUpBusWidth = 4, OutputDownBusWidth = 4
BlockSize = 4, BusCycle = 2
Arbitration_Scheme = BUS_ROUND_ROBIN
Coherence Protocol – Classical, WriteOnce, MESI, MOESI, Synapse, Berkeley, Illinois, Firefly and Dragon

NumberOfNodes – 8

D.4.2 Table of Results

<table>
<thead>
<tr>
<th>Coherence Protocol</th>
<th>Cache Size (Lines)</th>
<th>Cache Size (KB)</th>
<th>Average Hit Rate (%)</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical</td>
<td>64</td>
<td>1</td>
<td>88.90</td>
<td>25403201</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>95.89</td>
<td>14527839</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>99.13</td>
<td>9103579</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.50</td>
<td>8684834</td>
</tr>
<tr>
<td>Write-Once</td>
<td>64</td>
<td>1</td>
<td>92.97</td>
<td>24709777</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>96.22</td>
<td>11497217</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>99.23</td>
<td>4000614</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.66</td>
<td>3716655</td>
</tr>
<tr>
<td>MESI</td>
<td>64</td>
<td>1</td>
<td>92.97</td>
<td>24710086</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>96.22</td>
<td>11502895</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>99.23</td>
<td>4015966</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.66</td>
<td>3765519</td>
</tr>
<tr>
<td>MOESI</td>
<td>64</td>
<td>1</td>
<td>92.97</td>
<td>24707685</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>96.23</td>
<td>11650701</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>99.23</td>
<td>3982653</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.66</td>
<td>3665428</td>
</tr>
<tr>
<td>Synapse</td>
<td>64</td>
<td>1</td>
<td>92.97</td>
<td>24710434</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>96.43</td>
<td>13438924</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>99.19</td>
<td>4111675</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.60</td>
<td>3798630</td>
</tr>
<tr>
<td>Berkeley</td>
<td>64</td>
<td>1</td>
<td>92.97</td>
<td>24707068</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>96.12</td>
<td>11022333</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>99.23</td>
<td>3949449</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.66</td>
<td>3652679</td>
</tr>
<tr>
<td>Illinois</td>
<td>64</td>
<td>1</td>
<td>92.94</td>
<td>24015251</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>95.87</td>
<td>9533393</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>99.22</td>
<td>3866227</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.66</td>
<td>3680378</td>
</tr>
<tr>
<td>Firefly</td>
<td>64</td>
<td>1</td>
<td>92.94</td>
<td>24002207</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>96.06</td>
<td>9860150</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>99.41</td>
<td>6357985</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.83</td>
<td>8311403</td>
</tr>
<tr>
<td>Dragon</td>
<td>64</td>
<td>1</td>
<td>92.94</td>
<td>24017483</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>8</td>
<td>96.04</td>
<td>9882902</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>64</td>
<td>99.37</td>
<td>6078592</td>
</tr>
<tr>
<td></td>
<td>32768</td>
<td>512</td>
<td>99.81</td>
<td>7861821</td>
</tr>
</tbody>
</table>
D.5 Number of Processors in a Bus-Based Multiprocessor Experiment

D.5.1 Parameter Values

Processor
OutputDownBusWidth – 1
Matrix_Dimension – 128, Matrix_Block_Size – 8
Synchronisation – CentralBarrier
Memory_Conistency – Sequential_WeakOrder, Instruction_Buffer_Size – 1
LDI_delay – 1, SUB_delay – 1, ADD_delay – 1, MULT_delay – 3,
ADDD_delay – 2, SUBD_delay – 2, MULTD_delay – 6, DIVD_delay – 12,
UBR_delay – 1, CBR_delay – 1

Cache
OutputUpBusWidth – 1, OutputDownBusWidth – 4
BlockSize – 4, CacheLines – 1024
Allocation_Policy – WRITE_ALLOC, Write_Policy – COPY_BACK
Replacement_Policy – LRU, Associativity – 1
Coherence_Protocol – Berkeley or Firefly
Level – 1
ReadDelay – 1, WriteDelay – 1

Memory
OutputUpBusWidth – 4
BlockSize – 4
MemorySize – 40000
MemoryReadDelay – 5, MemoryWriteDelay – 5
Coherence_Protocol – Berkeley or Firefly

Bus
OutputUpBusWidth – 4, OutputDownBusWidth – 4
BlockSize – 4
BusCycle – 2  
Arbitration_Scheme – BUS_ROUND_ROBIN  
Coherence_Protocol – Berkeley or Firefly  
NumberOfNodes – 1, 2, 4, 8, 16 and 32  

D.5.2 Table of Results

<table>
<thead>
<tr>
<th>Coherence Protocol</th>
<th>Number of Processors</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley</td>
<td>1</td>
<td>20225279</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11510326</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6750405</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4819219</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4183109</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4016943</td>
</tr>
<tr>
<td>Firefly</td>
<td>1</td>
<td>19977419</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11661804</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6647211</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4999746</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4408171</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4386091</td>
</tr>
</tbody>
</table>
D.6 Different Processor Speeds Experiment

D.6.1 Parameter Values

Processor
OutputDownBusWidth – 1
Matrix_Dimension – 128, Matrix_Block_Size – 16
Synchronisation – CentralBarrier
Memory_Consistency – Sequential_WeakOrder, Instruction_Buffer_Size – 1
LDI_delay – 0,1 and 2, SUB_delay – 0, 1 and 2, ADD_delay – 0, 1 and 2,
MULT_delay – 0, 3 and 6, ADDD_delay – 0, 2 and 4, SUBD_delay – 0, 2 and 4,
MULTD_delay – 0, 6 and 12, DIVD_delay – 0, 12 and 24, UBR_delay – 0, 1 and 2,
CBR_delay – 0, 1 and 2

Cache
OutputUpBusWidth – 1, OutputDownBusWidth – 4
BlockSize – 4, CacheLines – 1024
Allocation_Policy – WRITE_ALLOC, Write_Policy – COPY_BACK
Replacement_Policy – LRU, Associativity – 1
Coherence_Protocol – Illinois
Level – 1, ReadDelay – 1, WriteDelay – 1

Memory
OutputUpBusWidth – 4
BlockSize – 4
MemorySize – 40000
MemoryReadDelay – 5, MemoryWriteDelay – 5
Coherence_Protocol – Illinois

Bus
OutputUpBusWidth – 4, OutputDownBusWidth – 4
BlockSize – 4, BusCycle – 2
Arbitration_Scheme – BUS_ROUND_ROBIN
Coherence_Protocol – Illinois
NumberOfNodes – 8
## D.6.2 Table of Results

<table>
<thead>
<tr>
<th>Instruction Category</th>
<th>Delay</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Arithmetic</td>
<td>0</td>
<td>4243311</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5469440</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>0</td>
<td>4788361</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4912691</td>
</tr>
<tr>
<td>DP Arithmetic</td>
<td>0</td>
<td>4580054</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5140839</td>
</tr>
<tr>
<td>DP Multiply</td>
<td>0</td>
<td>4120367</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>5623131</td>
</tr>
<tr>
<td>DP Divide</td>
<td>0</td>
<td>4794458</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>4919567</td>
</tr>
<tr>
<td>Branches</td>
<td>0</td>
<td>4571154</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5126631</td>
</tr>
<tr>
<td>All Categories</td>
<td>Minimum</td>
<td>3154066</td>
</tr>
<tr>
<td></td>
<td>Middle</td>
<td>4850933</td>
</tr>
<tr>
<td></td>
<td>Maximum</td>
<td>6986735</td>
</tr>
</tbody>
</table>
D.7 Synchronisation Primitive Implementation Experiment

D.7.1 Parameter Values

**Processor**
OutputDownBusWidth \(-1\)
Matrix_Dimension \(-128\), Matrix_Block_Size \(-16\)
Synchronisation – CentralBarrier, CentralBarrierBW and CentralBarrierOpt
Memory_Consistency – Sequential_WeakOrder, Instruction_Buffer_Size \(-1\)
LDI_delay \(-1\), SUB_delay \(-1\), ADD_delay \(-1\), MULT_delay \(-3\), ADDD_delay \(-2\), SUBD_delay \(-2\), MULTD_delay \(-6\), DIVD_delay \(-12\), UBR_delay \(-1\), CBR_delay \(-1\)

**Cache**
OutputUpBusWidth \(-1\), OutputDownBusWidth \(-4\)
BlockSize \(-4\), CacheLines \(-1024\)
Allocation_Policy – WRITE_ALLOC, Write_Policy – COPY_BACK
Replacement_Policy – LRU, Associativity \(-1\)
Coherence_Protocol – Berkeley and Firefly
Level \(-1\), ReadDelay \(-1\), WriteDelay \(-1\)

**Memory**
OutputUpBusWidth \(-4\)
BlockSize \(-4\)
MemorySize \(-40000\)
MemoryReadDelay \(-5\), MemoryWriteDelay \(-5\)
Coherence_Protocol – Berkeley and Firefly

**Bus**
OutputUpBusWidth \(-4\), OutputDownBusWidth \(-4\)
BlockSize \(-4\), BusCycle \(-2\)
Arbitration_Scheme – BUS_ROUND_ROBIN
Coherence_Protocol – Berkeley and Firefly
NumberOfNodes \(-8\)
### Table of Results

<table>
<thead>
<tr>
<th>Coherence Protocol</th>
<th>Synchronisation Implementation</th>
<th>Simulation Time</th>
<th>Simulation Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley</td>
<td>Shared-Memory</td>
<td>4971679</td>
<td>6606580</td>
</tr>
<tr>
<td></td>
<td>Private Busy-Waiting</td>
<td>4963684</td>
<td>3844177</td>
</tr>
<tr>
<td></td>
<td>Private Interrupts</td>
<td>4962681</td>
<td>2891718</td>
</tr>
<tr>
<td>Firefly</td>
<td>Shared-Memory</td>
<td>5602366</td>
<td>10611639</td>
</tr>
<tr>
<td></td>
<td>Private Busy-Waiting</td>
<td>5595621</td>
<td>4485150</td>
</tr>
<tr>
<td></td>
<td>Private Interrupts</td>
<td>5595618</td>
<td>3632012</td>
</tr>
</tbody>
</table>
D.8 Cache Size in a Clustered System

D.8.1 Parameter Values

Processor
OutputDownBusWidth = 1
Matrix_Dimension = 128, Matrix_Block_Size = 16
Synchronisation = CentralBarrierOpt
Memory_Consistency = Sequential_WeakOrder, Instruction_Buffer_Size = 1
LDI_delay = 1, SUB_delay = 1, ADD_delay = 1, MULT_delay = 3,
ADDD_delay = 2, SUBD_delay = 2, MULTD_delay = 6, DIVD_delay = 12,
UBR_delay = 1, CBR_delay = 1

Cache
OutputUpBusWidth = 1, OutputDownBusWidth = 4
BlockSize = 4, CacheLines = 32, 128, 512, 2048, 8192 and 32768
Allocation_Policy = WRITE_ALLOC, Write_Policy = COPY_BACK
Replacement_Policy = LRU, Associativity = 1
Coherence_Protocol = Berkeley
Level = 1, ReadDelay = 1, WriteDelay = 1

Memory
OutputUpBusWidth = 4
BlockSize = 4
MemorySize = 10000
MemoryReadDelay = 5, MemoryWriteDelay = 5
Coherence_Protocol = Berkeley

Bus
OutputUpBusWidth = 4, OutputDownBusWidth = 4
BlockSize = 4, BusCycle = 2
Arbitration_Scheme = BUS_ROUND_ROBIN
Coherence_Protocol = Berkeley
NumberOfNodes = 4
Lower Level Receiver
OutputUpBusWidth – 4, OutputDownBusWidth – 4
BlockSize – 4
Coherence_Protocol – Berkeley

Lower Level Sender
OutputUpBusWidth – 4, OutputDownBusWidth – 4
BlockSize – 4
Coherence_Protocol – Berkeley

Upper Level Receiver
OutputUpBusWidth – 4, OutputDownBusWidth – 4
BlockSize – 4
Coherence_Protocol – FullMap

Upper Level Sender
OutputUpBusWidth – 4, OutputDownBusWidth – 4
BlockSize – 4
Coherence_Protocol – FullMap

Interconnection Network
Type – Crossbar
Delay – 2
Number Of Nodes – 4

D.8.2 Table of Result

<table>
<thead>
<tr>
<th>Cache Size (Lines)</th>
<th>Cache Size (KB)</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.5</td>
<td>17041342</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td>10708702</td>
</tr>
<tr>
<td>512</td>
<td>8</td>
<td>5325177</td>
</tr>
<tr>
<td>2048</td>
<td>32</td>
<td>2799369</td>
</tr>
<tr>
<td>8192</td>
<td>128</td>
<td>2520188</td>
</tr>
<tr>
<td>32768</td>
<td>512</td>
<td>2515679</td>
</tr>
</tbody>
</table>
D.9 Cluster Configuration Experiment

D.9.1 Parameter Values

**Processor**
OutputDownBusWidth = 1
Matrix_Dimension = 128, Matrix_Block_Size = 8
Synchronisation – CentralBarrierOpt
Memory_Consistency – Sequential_WeakOrder, Instruction_Buffer_Size = 1
LDI_delay = 1, SUB_delay = 1, ADD_delay = 1, MULT_delay = 3,
ADDD_delay = 2, SUBD_delay = 2, MULTD_delay = 6, DIVD_delay = 12,
UBR_delay = 1, CBR_delay = 1

**Cache**
OutputUpBusWidth = 1, OutputDownBusWidth = 4
BlockSize = 4, CacheLines = 1024
Allocation_Policy – WRITE_ALLOC, Write_Policy – COPY_BACK
Replacement_Policy – LRU, Associativity – 1
Coherence_Protocol – Berkeley
Level – 1, ReadDelay – 1, WriteDelay – 1

**Memory**
OutputUpBusWidth = 4
BlockSize = 4
MemorySize = 10000
MemoryReadDelay = 5, MemoryWriteDelay = 5
Coherence_Protocol – Berkeley

**Bus**
OutputUpBusWidth = 4, OutputDownBusWidth = 4
BlockSize = 4, BusCycle = 2
Arbitration_Scheme – BUS_ROUND_ROBIN
Coherence_Protocol – Berkeley
NumberOfNodes = 1, 2, 4, 8, 16 and 32
Lower Level Receiver
OutputUpBusWidth – 4, OutputDownBusWidth – 4
BlockSize – 4
Coherence_Protocol – Berkeley

Lower Level Sender
OutputUpBusWidth – 4, OutputDownBusWidth – 4
BlockSize – 4
Coherence_Protocol – Berkeley

Upper Level Receiver
OutputUpBusWidth – 4, OutputDownBusWidth – 4
BlockSize – 4
Coherence_Protocol – FullMap

Upper Level Sender
OutputUpBusWidth – 4, OutputDownBusWidth – 4
BlockSize – 4
Coherence_Protocol – FullMap

Interconnection Network
Type – Crossbar
Delay – 2, 10 and 18
Number Of Nodes – 1, 2, 4, 8, 16 and 32
### D.9.2 Table of Results

<table>
<thead>
<tr>
<th>Interconnection Network Delay</th>
<th>Number of Clusters</th>
<th>Number of Processors Per</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>32</td>
<td>4016943</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>2823448</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>2083390</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>1742244</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2</td>
<td>1690449</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>1697650</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>32</td>
<td>4016943</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>2896711</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>2205903</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>1888601</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2</td>
<td>1837232</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>1857726</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>32</td>
<td>4016943</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>2943002</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>2328922</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>2034958</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2</td>
<td>1982138</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>2017802</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>32</td>
<td>4016943</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>2999344</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>2437865</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>2192657</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2</td>
<td>2140498</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1</td>
<td>2191671</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>32</td>
<td>4016943</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>3033661</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>2500177</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>2362577</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2</td>
<td>2311510</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2</td>
<td>2380522</td>
</tr>
</tbody>
</table>