ADAPTIVE JAVA OPTIMISATION USING MACHINE LEARNING TECHNIQUES

Shun Long

Doctor of Philosophy
University of Edinburgh
2004
Dedicated to my father, Ni Long
Abstract

There is a continuing demand for higher performance, particularly in the area of scientific and engineering computation. In order to achieve high performance in the context of frequent hardware upgrading, software must be adaptable for portable performance. What is required is an optimising compiler that evolves and adapts itself to environmental change without sacrificing performance.

Java has emerged as a dominant programming language widely used in a variety of application areas. However, its architectural independent design means that it is frequently unable to deliver high performance especially when compared to other imperative languages such as Fortran and C/C++.

This thesis presents a language- and architecture-independent approach to achieve portable high performance. It uses the mapping notation introduced in the Unified Transformation Framework[65] to specify a large optimisation space. A heuristic random search algorithm is introduced to explore this space in a feedback-directed iterative optimisation manner. It is then extended using a machine learning approach which enables the compiler to learn from its previous optimisations and apply the knowledge when necessary.

Both the heuristic random search algorithm and the learning optimisation approach are implemented in a prototype Adaptive Optimisation Framework for Java (AOF-Java). The experimental results show that the heuristic random search algorithm can find, within a relatively small number of attempts, good points in the large optimisation space. In addition, the learning optimisation approach is capable of finding good transformations for a given program from its prior experience with other similar programs.
Acknowledgements

I would like to thank my supervisor, Michael O’Boyle, not only for his guidance, patience and encouragement over the past four years, but also for showing me the art of research via countless discussions.

I would like to thank my Mum and Dad, for their constant love, support and encouragement, for helping me to become the man I want to be, and for guiding me all along the way to PhD study. Thanks a lot to my sister Li and her partner Willy. Without their help and support, I would not have the chance to study abroad.

Special thanks to Kennis Lau, for her making my time in Edinburgh one of the most enjoyable parts of my life to date, for her endless support, understanding and encouragement since the day we knew each other, and for her always sharing my high and low.

Thanks a lot to Tim Jones, Ben Kavanagh and Grigori Fursin in Edinburgh, as well as Peter Knijnenburg in Netherland, for kindly reading the drafts and correcting my mistakes. Thanks also to Mark Bull, Marcelo Cintra, Bjoern Franke, Tom Ashby, and JiaLin Dou for our discussions all these years.

Special thanks to Chengdian Cai, my former adviser in China, who gave me my first taste and lesson of research during my university years and afterward.

Thanks a lot to my aunt, my cousin Jack Wong and his family (Yvonne and Christie) for giving me warm and happy Christmas.

Thanks also to all of my friends and former colleagues who keep encouraging me all these years, as well as to all the friends I have here who made my dormitory life a brilliant experience to enjoy: Andy, SiJie, George, Paolo, Rogers, Baba, Brian, Jose, Christian, Catherine, Lena, Argyro, Sophie, HouKiat, Pablo, Vincent, John, Will, Roland, Audrey, Lech and Massimo.
Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(Shun Long)
# Table of Contents

List of Figures 5

Chapter 1 Introduction 9
  1.1 Motivation 9
  1.2 AOF-Java 11
  1.3 Contribution 12
  1.4 Outline 13

Chapter 2 Related Work 14
  2.1 Introduction 14
  2.2 Java 14
    2.2.1 Java as a programming language 14
    2.2.2 Java Virtual Machine 15
  2.3 Java Optimisation 17
    2.3.1 Static optimisation 18
    2.3.2 Just-in-time compilation 19
    2.3.3 JVM optimisation 21
    2.3.4 Array optimisation 22
  2.4 Adaptive Optimization 22
    2.4.1 Introduction 22
    2.4.2 Adaptive optimisation for Java 24
    2.4.3 Non-Java adaptive optimisation 25
    2.4.4 Summary 27
  2.5 Iterative Optimisation 28
    2.5.1 Concepts and model 28
    2.5.2 Search in parameterised optimisation spaces 30
    2.5.3 Search in non-parameterised optimisation spaces 33
    2.5.4 Lifelong program analysis and transformation 34
  2.6 Systematic Exploration of Optimisation Space 35
    2.6.1 Introduction 35
2.6.2 Mapping construction based on performance estimation . . 36
2.6.3 Genetic Algorithm Parallelism System .......................... 36
2.7 Optimisations Using Machine-Learning Approaches .......... 38
2.7.1 Introduction ..................................................... 38
2.7.2 Case-based reasoning for interactive performance tuning . 39
2.7.3 Classification based on statistical group performance . . 39
2.7.4 Genetic algorithm for DSP optimisation ..................... 40
2.7.5 Genetic programming for compiler heuristic optimisation . 41
2.7.6 Machine learning for instruction scheduling ............... 41
2.8 Summary ......................................................... 42

Chapter 3 Search Space .............................................. 43
3.1 Introduction ..................................................... 43
3.2 The Search Problem ............................................. 43
3.3 Unified Transformation Framework (UTF) ..................... 46
  3.3.1 Mapping ....................................................... 46
  3.3.2 Dependence ................................................... 48
  3.3.3 Legality test ................................................. 49
  3.3.4 Supporting facilities ........................................ 49
3.4 The Search Space ............................................... 50
  3.4.1 Properties of UTF ........................................... 50
  3.4.2 A closer look ............................................... 52
  3.4.3 A naive exhaustive scan algorithm ....................... 53
3.5 Features and Extension ......................................... 55
3.6 Summary ......................................................... 56

Chapter 4 Heuristic Search ........................................... 57
4.1 Introduction ..................................................... 57
4.2 Search Strategy .................................................. 57
  4.2.1 Additional notations ......................................... 58
  4.2.2 Mapping construction as a two phase process .......... 59
  4.2.3 Random ....................................................... 59
  4.2.4 Rotation policy .............................................. 59
  4.2.5 Loop first, then syntactic ................................ 60
  4.2.6 Simple first .................................................. 60
  4.2.7 Explore to depth if a good hint is found ............... 61
4.3 Heuristic Search Algorithm ..................................... 61
  4.3.1 Bias mechanism ............................................. 61

2
4.3.2 Supporting facilities .............................................. 62
4.3.3 The algorithm ..................................................... 63
4.4 L-Search ............................................................. 64
  4.4.1 Introduction .................................................... 64
  4.4.2 Construct a loop vector ...................................... 65
  4.4.3 Generate the default syntactic matrix .................... 73
  4.4.4 Construct the default schedule ............................ 75
  4.4.5 Explore to further depth .................................... 75
  4.4.6 Bias in L-Search control ................................... 79
4.5 S-Search ............................................................. 79
  4.5.1 Choose loop vectors ......................................... 80
  4.5.2 Syntactic matrix generation ............................... 81
  4.5.3 Construct the schedule ..................................... 86
  4.5.4 Another approach to choose loop vector(s) ............ 86
  4.5.5 Bias in S-Search control ................................... 87
4.6 Coverage ............................................................ 87
  4.6.1 Redundancy check .......................................... 87
  4.6.2 Coverage of the algorithm ................................ 88
4.7 Comparison ........................................................ 89
4.8 Summary ............................................................ 92

Chapter 5 Learning Optimisation ..................................... 93
  5.1 Introduction ...................................................... 93
  5.2 Learning Compiler .............................................. 93
    5.2.1 Concept ...................................................... 93
    5.2.2 System Objectives ......................................... 94
    5.2.3 Existing machine learning approaches ................ 95
    5.2.4 Decision .................................................... 98
  5.3 Instance-based Learning Optimisation (IBLO) ............. 98
    5.3.1 Basic ideas ............................................... 98
    5.3.2 Classification .............................................. 100
    5.3.3 Knowledge storage ....................................... 103
    5.3.4 Transformation selection ................................ 104
  5.4 Comparison ...................................................... 107
  5.5 Summary ........................................................ 109
List of Figures

3.1 Execution time vs tiling .................................................. 44
3.2 Execution time vs unrolling ............................................ 45
3.3 A simple loop example and its identity mappings ................. 47
3.4 The loop after applying the mapping which swaps the last syntactic components ........................................... 47
3.5 The loop after applying the mapping which hoists the last syntactic components to a higher level .................. 48
3.6 Example of loop tiling ..................................................... 51
3.7 Different schedules representing the same transformation ...... 54

4.1 An example of default schedule ........................................ 59
4.2 Pseudo code of the steering module ................................... 64
4.3 Pseudo code for deciding tile sizes .................................... 66
4.4 Code after loop tiling ...................................................... 67
4.5 Pseudo code for deciding unrolling factor(s) ...................... 68
4.6 Code after loop unrolling ................................................ 68
4.7 A simple matrix (left) and a less-simple matrix (right). The left matrix is considered simpler because it can be obtained by applying fewer steps of linear transformations to the starting matrix. Intuitively, it contains more 0s than the right one. .......... 70
4.8 An example of a default transformation matrix ................. 71
4.9 Pseudo code for generating a random simple transformation matrix ................................................................. 71
4.10 Pseudo code of the loop vector construction ..................... 72
4.11 Pseudo code for generating the default syntactic matrix ...... 74
4.12 Example of a default syntactic matrix and the resulting code . 76
4.13 Pseudo code of the default L-Search .............................. 77
4.14 Pseudo code for identifying "significantly good" loop vector ...... 79
4.15 Pseudo code of the to-depth L-Search ............................. 80
4.16 Syntactic matrix’s impact on resulting code (example.1) .... 81
4.17 Syntactic matrix’s impact on resulting code (example.2) .... 82
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.18</td>
<td>Syntactic matrix's impact on resulting code (example.3)</td>
<td>82</td>
</tr>
<tr>
<td>4.19</td>
<td>Syntactic matrix's impact on resulting code (example.4)</td>
<td>83</td>
</tr>
<tr>
<td>4.20</td>
<td>Pseudo code of the new syntactic matrix generation</td>
<td>85</td>
</tr>
<tr>
<td>4.21</td>
<td>Pseudo code of the default S-Search</td>
<td>86</td>
</tr>
<tr>
<td>4.22</td>
<td>Pseudo code of the non-default S-Search</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Pseudo code of the transformation selection</td>
<td>106</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of the transformation selection algorithm</td>
<td>108</td>
</tr>
<tr>
<td>6.1</td>
<td>Architecture of AOF-Java</td>
<td>111</td>
</tr>
<tr>
<td>6.2</td>
<td>Code segment with ProfilerStub</td>
<td>114</td>
</tr>
<tr>
<td>6.3</td>
<td>Summary of Benchmarks Used</td>
<td>115</td>
</tr>
<tr>
<td>6.4</td>
<td>Heuristic search on Linux+Celeron summary</td>
<td>117</td>
</tr>
<tr>
<td>6.5</td>
<td>The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel3 (upper) and kernel5 (lower) on Linux+Celeron. The standard deviations are shown in the error bars.</td>
<td>118</td>
</tr>
<tr>
<td>6.6</td>
<td>The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel6 (upper) and kernel7 (lower) on Linux+Celeron. The standard deviations are shown in the error bars.</td>
<td>119</td>
</tr>
<tr>
<td>6.7</td>
<td>The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel8 (upper) and kernel9 (lower) on Linux+Celeron. The standard deviations are shown in the error bars.</td>
<td>120</td>
</tr>
<tr>
<td>6.8</td>
<td>The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel10 (upper) and kernel11 (lower) on Linux+Celeron. The standard deviations are shown in the error bars.</td>
<td>121</td>
</tr>
<tr>
<td>6.9</td>
<td>The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel12 (upper) and kernel19 (lower) on Linux+Celeron. The standard deviations are shown in the error bars.</td>
<td>122</td>
</tr>
<tr>
<td>6.10</td>
<td>The solid curves show the average of the best execution time found so far during the ten heuristic searches on mm (upper) and dol- teration (lower) on Linux+Celeron. The standard deviations are shown in the error bars.</td>
<td>123</td>
</tr>
</tbody>
</table>
6.11 The solid curves show the average of the best execution time found so far during the ten heuristic searches on runF (upper) and runG (lower) on Linux+Celeron. The standard deviations are shown in the error bars. .................................................. 124

6.12 The solid curves show the average of the best execution time found so far during the ten heuristic searches on runR (upper) and runS (lower) on Linux+Celeron. The standard deviations are shown in the error bars. .................................................. 125

6.13 The curve demonstrates one heuristic search on kernel6 on Linux +Celeron. The execution time is plotted against the number of evaluations. The large variation in kernel6's performance caused by different transformations demonstrates the complexity of the optimisation space. .................................................. 127

6.14 Heuristic search on Windows+PentiumPro summary ............ 128

6.15 The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel3 (upper) and kernel5 (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars. .................................................. 130

6.16 The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel6 (upper) and kernel7 (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars. .................................................. 131

6.17 The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel8 (upper) and kernel9 (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars. .................................................. 132

6.18 The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel10 (upper) and kernel11 (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars. .................................................. 133

6.19 The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel12 (upper) and kernel19 (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars. .................................................. 134
6.20 The solid curves show the average of the best execution time found
so far during the ten heuristic searches on mm (upper) and doIter-
ation (lower) on Windows+PentiumPro. The standard deviations
are shown in the error bars. .................................................. 135

6.21 The solid curves show the average of the best execution time found
so far during the ten heuristic searches on runF (upper) and runG
(lower) on Windows+PentiumPro. The standard deviations are
shown in the error bars. .................................................. 136

6.22 The solid curves show the average of the best execution time found
so far during the ten heuristic searches on runR (upper) and runS
(lower) on Windows+PentiumPro. The standard deviations are
shown in the error bars. .................................................. 137

6.23 The curve demonstrates one heuristic search on kernel9 on Win-
dows+PentiumPro. The execution time is plotted against the num-
ber of evaluations. The negligible variation in kernel9's perform-
ance shows that its performance is not affected by the transfor-
mations applied. .................................................. 138

6.24 Program features of benchmarks (before pre-processing) ......... 140

6.25 Program features of benchmarks (after pre-processing) ........... 141

6.26 Categories, benchmarks and the most similar categories .......... 141

6.27 Benchmarks and their transformation sources ......................... 142

6.28 Results of applying the chosen transformations to the benchmarks
on Linux+Celeron .................................................. 143

6.29 Comparison between results of the heuristic search and of learning
on Linux+Celeron .................................................. 144

6.30 Results of applying the chosen transformations to the benchmarks
on Windows+PentiumPro .................................................. 147

6.31 Comparison between results of the heuristic search and of learning
on Windows+PentiumPro .................................................. 148
Chapter 1

Introduction

1.1 Motivation

The rapid growth in the use of computer technology has enlarged the range of application areas and generated a demand for higher performance machines. This is particularly true in scientific and engineering computation, which is characterised by intensive numerical computing. Much research\cite{2,26,96,104,129,130} has been undertaken to develop parallelisation techniques for different architectures in order to achieve high performance in these areas. Compared to the human costs, hardware was relatively expensive in the past, so it was cost-effective to spend considerable efforts carefully tuning a specific application for a specific platform. It was expected to tune for a considerable amount of time before the hardware was upgraded and hence little concern was given to portability.

As the evolution of hardware and architecture becomes faster\cite{126}, new and cheaper hardware appears with higher performance. Parallel, distributed and even heterogeneous network computing are now not only feasible but also widely used, and grid computing\cite{49} is emerging as a new computation paradigm. Considering the frequency of hardware upgrading and the fact that human costs now dominate, it is no longer cost-effective if each hardware upgrade means the application has to be re-tuned from scratch. In a network computing environment this is particularly true, as it is not feasible to tune the application from scratch every time a new node of different architecture joins the network. In order to achieve high performance in modern computing environments in the context of frequent hardware upgrading, software is expected to be adaptable to hardware in order to make the high performance portable. One possible way to adapt software to hardware is via techniques such as adaptive compilation\cite{10,121}, iterative optimisation\cite{70} and speculative execution\cite{83}.

Considering the complexity of a modern computing environment and the rapid
change of hardware and architecture, portability should be given higher priority during software development in order to achieve high performance in an easier and long-term cost-effective manner.

Language selection plays an important part in achieving both performance and portability. Some languages are highly portable, others not, and likewise some can give high performance at the expense of other features (automatic memory management, for example). Java[53] has emerged as a dominant programming language, widely used in a variety of application areas from embedded systems to those in distributed enterprise-wide information processing. Besides it being simple, object-oriented, robust and secure, Java has other useful and important features. First, Java is architecture-neutral and portable. Java applications can run on any system with Java Virtual Machine (JVM) support. This feature makes the environmental variety of heterogeneous networks transparent and the "write once, run anywhere" philosophy possible. Developers do not need to consider the underlying environment, as long as it supports JVM. Secondly, Java is distributed. With its rich libraries of high-level networking support, it is easier to develop network applications in modern computing environments. Finally, Java’s support for multi-threading enables efficient parallelisation which helps to achieve high performance. These features make Java appealing for scientific and engineering computation in a modern network environment. However, high and portable performance must be achieved to make Java widely acceptable in this area dominated by other programming languages.

Unfortunately, the runtime performance of many Java applications is worse than those written in traditional compile-link-run languages such as Fortran and C/C++. This is due to the interpretation-execution mechanism of Java bytecode and/or other factors such as its garbage collection and stack-based memory model[112]. Java Grande Forum[61] has identified five critical Java language and virtual machine issues related to Java’s applicability to scientific and engineering computation. They are lack of support for complex arithmetic, lack of support for lightweight classes, no operator overloading, inability to use available floating point hardware, and inefficient multidimensional arrays. Many optimisation approaches have been developed in order to improve Java’s runtime performance.

Traditional compilation technologies are based on static analysis and a hard-wired compilation strategy. Its lack of adaptability to the runtime environment has been partly solved by the introduction of new optimisation techniques such as adaptive optimisation[10][121], iterative optimisation[70] and speculative execution[83]. These techniques adopt more flexible approaches and/or take
advantage of runtime information. Previous work demonstrates that they can improve application performance significantly.

To a certain extent, these optimisation approaches can be thought of as an automation of the approach which software developers use to tune their products before shipping in order to meet performance requirements. They are application-oriented and environment-specific. When the compiler meets another program, it has to start the whole optimisation process again from scratch because little, if any, mechanism is provided for it to take advantage of its prior optimisation knowledge. On the other hand, the rapid evolution of hardware and the appearance of novel architectures make it more complex for compilation to exploit its full potential. It is unrealistic to expect applications compiled with these approaches to be able to deliver good and portable performance. The key to this problem is the compiler's lack of ability to accumulate knowledge from its previous optimisation efforts and use it in the future, either in the same or a new environment.

Ideally, an optimising compiler should evolve and adapt itself to environment change without sacrificing performance.

1.2 AOF-Java

This thesis describes an Adaptive Optimisation Framework for Java (AOF-Java) in response to the problems described in the previous section. It is a source-to-source Java optimisation framework aiming to provide continuous performance improvement to Java applications via feedback-directed iterative optimisation and machine learning techniques.

AOF-Java uses the mapping notation presented by Unified Transformation Framework (UTF)[65] to specify an optimisation space consisting of a large collection of iteration reordering transformations. This space is large, highly irregular in shape, and highly non-linear with many local minima, mainly due to the interference among different transformations. AOF-Java explores this space using a heuristic search algorithm. It translates each point the search algorithm reaches into a transformation sequence and applies this sequence to the given program. Profiles are recorded and used as feedback to direct the heuristic search. All the above steps are carried out in an iterative manner until the optimisation budget is reached. Experimental results show that this heuristic search algorithm can achieve good performance improvement.

Furthermore, AOF-Java collects all the above optimisation profiles and uses a
machine learning approach to accumulate its knowledge of program optimisation from these previous attempts. The prior knowledge will later be applied to optimise new programs. Experimental results demonstrate that the prior knowledge is indeed useful for optimisation of other programs.

In brief, AOF-Java is capable of optimising Java programs from scratch, learning from these optimisation efforts, and applying its prior knowledge when necessary.

1.3 Contribution

The main contributions of this thesis are summarised as below.

The heuristic search algorithm and learning optimisation approach are portable because all the transformations considered are environment-independent and applied in a source-to-source manner. Portability is a crucial concern for modern computing environments as they tend to be distributed in order to use as many available network resources as possible. The search and learning approaches presented in this thesis are language-, implementation- and architecture-independent and therefore highly portable.

It is among the first to introduce machine learning techniques to an optimising compiler. There has been some similar work, as summarised in the Related Work chapter, but the problems they consider are specific, for example tuning an instruction scheduler, choosing optimal loop unrolling factor, or searching for an optimal implementation for a specific digital signal processing algorithm. In this thesis, machine learning techniques are used to steer the optimising compiler at a general and high level, and the prior knowledge it obtains from previous optimisations is explicitly applied when necessary.

It studies the impact of high-order transformations on Java optimisation. While other Java optimisation projects usually apply light-weight optimisations at the intermediate representation or bytecode level, it demonstrates that high-order transformations can bring significant performance improvement to Java, if used properly.

It considers a much larger optimisation space than previous projects. This is because UTF provides a unified representation of iteration reordering transformations. Any arbitrary combination of these transformations that can be represented by mapping is included into the space this thesis considers. Much previous work in iterative optimisation considered a very limited space with just one or a few transformations and usually in a fixed phase order manner.
In brief, this thesis successfully integrates high-order transformation, feedback-directed optimisation, iterative optimisation and machine learning techniques in order to achieve high and portable performance for Java applications.

1.4 Outline

This first chapter of the thesis gives a brief introduction to the background and contribution of the research. Related work is summarised in greater depth in Chapter 2, including brief reviews of the key technologies and previous work.

Chapter 3 specifies the problem to tackle and demonstrates its complexity by way of an example. It gives a brief review of the Unified Transformation Framework (UTF)[65], specifies a UTF-based optimisation space, and presents a naive exhaustive scan algorithm to search the entire space. An example is given to demonstrate the size of this search space.

Chapter 4 presents a heuristic search algorithm based on the above optimisation space, each step of which is specified in depth. It briefly discusses the coverage of this search algorithm before making a comparison between this algorithm and others presented in previous work.

Chapter 5 introduces the concept of a learning compiler before presenting an instance-based learning optimisation approach.

Chapter 6 gives a brief introduction to the prototype AOF-Java framework. Experimental results are then presented in order to demonstrate the ability of both the heuristic search algorithm and the learning optimisation approach. These results demonstrate the potential of integrating proper machine learning techniques into an optimising compiler.

Chapter 7 presents a summary of this thesis and a critique of the work, before outlining future work.
Chapter 2

Related Work

2.1 Introduction

This thesis aims to develop an architecture- and environment-independent optimisation framework in order to achieve portable and continuous performance improvement in modern computing environments, especially for Java applications in the area of scientific and engineering computation. This framework adaptively optimises Java applications in an iterative manner, using both a heuristic search algorithm and a machine learning approach based on Pugh's Unified Transformation Framework (UTF)[65].

This chapter briefly reviews related work in the areas of Java technology, Java optimisation, adaptive and iterative optimisation. It also reviews work based on UTF and optimisations using machine learning approaches.

2.2 Java

This section gives a brief review of Java technology, discussing both language and environment aspects.

2.2.1 Java as a programming language

As a programming language, Java[53] is simple, object-oriented, interpreted, dynamic, secure, robust and of reasonable performance. These features make it widely used in a range of application areas. In addition, Java has the following features which make it a favourable programming language for parallel, distributed and even heterogeneous network computing application development.

Distributed Java provides high-level support for network programming with its class libraries.
**Architecture-neutral and portable** The stack-based Java virtual machine is simple and general enough that it can be implemented on most modern processors. A Java application in bytecode can run on any system supporting the Java virtual machine. Java bytecode is therefore portable and conforms to the "write once, run anywhere" philosophy. This feature is of particular significance when considering the heterogeneous nature of the Internet and other modern computing environments.

**Multi-threaded** Java supports multiple threads of execution that can concurrently handle different tasks. Programmers can develop multi-threaded Java applications using the keyword "synchronized" and the class libraries.

Java provides a rich collection of classes[48], in various packages, that programmers can use in software development. The class libraries provide convenient support for a wide range of programming tasks, from text input/output and graphical user interfaces to network programming and multi-threading. Third-party class libraries can also be used to further extend Java's applicability in various domains.

The Java compiler compiles Java source into architecture-neutral bytecode and stores it in class files. Class files contain JVM instructions, the symbol table, and other auxiliary information covering all the information from the original Java source code. A description of the content of a class file can be found in [110], and its specification can be found in [78].

### 2.2.2 Java Virtual Machine

The Java virtual machine (JVM)[78] is an abstract machine which provides a set of basic operations based on a simple, stack-based operation model. This "no more than fundamental" design makes it possible that a JVM be implemented on a wide range of processors ranging from those used in embedded systems to those used in supercomputers. This contributes to the architectural neutrality of Java technology. On the other hand, although designed to be simple, JVM still imposes a strong format and structural constraints on the bytecode for security purposes.

#### 2.2.2.1 Data types

JVM operates on two kinds of types. One is a primitive type which includes integral, floating point, boolean and *return Address*. The other is a reference type which includes class, array and interface.
2.2.2.2 Runtime data areas

During program execution, the JVM allocates data areas for various purposes, which include program counter register, JVM stacks, heap, method area and optional native method stacks.

Each JVM thread has its private JVM stack which is used to keep both local variables and partial results and to help method invocation and return. Each method invocation is allocated a frame in the stack during the execution of the thread.

Heap is the runtime data area allocated for all class instances and arrays. It is created during the launch of JVM and shared among all JVM threads. Heap is managed by an automatic memory management system called a garbage collector.

Like heap, method area is also shared among all JVM threads. Each class stores, in its method area, its structures such as fields and method data, code for methods and constructors, and a runtime constant pool.

2.2.2.3 Frame

The JVM stack consists of frames, each of which corresponds to a method invocation. During the execution of a thread, a frame is created and allocated from the JVM stack when a method invocation occurs, and is destroyed when the invocation finishes. The frame is responsible for the following four tasks: storing data and partial results of the method, keeping parameters and results of method invocation, performing dynamic linking, and dispatching exceptions. Due to safety concerns, the frame is not accessible to any other thread during its lifetime.

The kernel component of the frame is its last-in-first-out operand stack, where the computation fetches the operands from and stores the result into. It is also used to prepare parameters to be passed to methods and to receive method results. This operand stack is the reason why JVM is considered stack-based. It is simple and therefore can be implemented on most processors available. This helps not only to achieve Java's architecture independence and portability but also to improve the code density of Java bytecode.

2.2.2.4 Support for class libraries

Sufficient support must be provided by the JVM for the implementation of the class libraries on the associated platform, especially for classes about creation and loading of a class/interface, linking and initialisation of a class/interface, security and multi-threading, etc.
2.2.2.5 JVM at program execution

By invoking the method main of the class, JVM starts execution by loading the class, linking it with other classes it uses, and initialising it.

Class loading is responsible for finding the binary form (normally in class file format) of the class/interface with the given name. It is carried out by either the class ClassLoader or its subclasses which implement different loading policies.

Linking takes a binary form of a class/interface and combines it into the runtime state of the JVM, so that it can be executed. It is responsible for carrying out the syntactic and semantic checks, allocating static storage and data structures for execution, and resolving symbolic references from one class/interface to the others.

When an object is created, JVM allocates memory space for all of its fields declared in its data type, as well as in all of its superclasses in the class hierarchy. When all the non-daemon threads terminate, or some thread invokes either Runtime.exit() or System.exit(), JVM terminates all its activities and exits.

2.2.2.6 Exception

When the method invocation completes normally, a value may be returned to the invoker method via the operand stack when the return instruction of the invoked method is executed. The frame of the invoker is then resumed in order to restore its state.

Method invocation may complete abruptly when an exception is thrown by the JVM during the execution of an instruction within the method but not handled within it, or when an athrow instruction is executed to explicitly throw an exception, and it is not caught by the current method. The exception-handling mechanism of JVM is explained in [78].

2.3 Java Optimisation

Java optimisation can be achieved via two approaches: through optimising compilation techniques and by providing an efficient JVM on which Java bytecode can run faster. There have been two major categories in the former approach: one tries static optimisation, and the other uses just-in-time compilation. These two categories are discussed in depth in the following subsections, before the efficient JVM approach is discussed.
2.3.1 Static optimisation

There have been two successful approaches to Java static optimisation. One automatically detects parallel code segments in a Java program and generates parallel code from the analysis. The projects javar\cite{21}, javab\cite{25}, JPT\cite{20} and JavaSpMT\cite{63} fall into this category. The second approach extends or modifies Java so that parallelism can be expressed explicitly. The projects Titanium\cite{132}, HPJava\cite{34}, JPVM\cite{47}, Java/DSM\cite{133} and Spar\cite{109} use this approach.

A good review of existing automatic parallelisation techniques is given in \cite{26}. The key techniques include data dependency analysis, induction and reduction variable recognition, array privatisation, run-time analysis (such as speculative run-time parallelisation) and others. These techniques have been widely used in high-performance computing areas previously dominated by Fortran and C/C++. Listed below are some projects which successfully apply these techniques to Java optimisation.

The javar project\cite{21}\cite{22}\cite{23}\cite{24} identifies and annotates parallel loops and/or multi-way recursive method calls in Java programs, and then exploits the potential parallelism by multi-threading. The javab project\cite{25} is similar to javar but implements automatic detection and exploitation of implicit loop parallelism at Java bytecode level. Experimental results show that speedup can be obtained on any JVM supporting true parallel execution of threads, and the overhead remains low on uni-processors.

Java Parallelization Tool (JPT)\cite{20} is a tool used to generate PVM (Parallel Virtual Machine)\cite{108} code from a serial Java program. It automatically detects parallel loops and generates master and slave PVM programs. The PVM library is needed to support parallel execution.

Java Speculative MultiThreading (JavaSpMT)\cite{63} improves Java performance by exploiting coarse-grained parallelism on a shared-memory multiprocessor system. It speculatively parallelises potentially dependent Java loops by source-to-source transformations, and executes them in a pipelined fashion with the help of control speculation, runtime dependency checking and multithreading.

Due to the complexity of automatic parallelisation, and to the fact that parallelisation tools were mainly developed for other languages such as Fortran and C/C++, many projects choose not to detect parallelism themselves. They either make use of existing results from previous successful projects or rely on the programmer to annotate parallelism explicitly.

Titanium\cite{132} is a Java-like programming language and compilation system designed for high-performance scientific computing. It is based on an SPMD
model of parallel computation and achieves parallelism via global synchronisation (explicit barrier), global/local references and special communication methods. The Titanium compiler turns Titanium source into C and therefore can take advantage of existing analysis and optimisation techniques widely used in C/C++.

HPJava[34] is an extension to Java based on SIMD parallel paradigm. The extension includes parallel arrays, true multi-dimensional arrays and process arrays. Distributed control constructs, such as the data-parallel loop construct, are also provided.

Similar to Titanium and HPJava, JPVM[47], Java/DSM[133] and Spar[109] are all Java dialects which extend or modify Java by using special data/code constructs and directives to explicitly annotate parallel code. The parallel execution relies on support from external systems/libraries such as PVM or Treadmarks DSM (Distributed Shared Memory)[64]. Although these modifications make programming in the corresponding paradigms easier, their difference from standard Java and their need for external support make it difficult for any of them to become a widely-accepted standard approach to Java optimisation.

Hyperion[7] compiles multi-threaded Java bytecode to native code. With the help of a runtime library, it runs the Java applications on a cluster of processors under a single Java machine image.

Almost-whole-program compilation framework[31] is another static optimisation technique. It transforms a collection of Java classes into a package, and applies source-to-source whole-program optimisations to all the source codes involved. This approach utilises the fact that the programs to be optimised are almost finalised and therefore allow optimisations to be carried out at an almost-global level. Experimental results show that this approach can balance Java flexibility and performance penalties.

### 2.3.2 Just-in-time compilation

The mechanism of just-in-time (JIT) compilation is to compile Java bytecode into native code immediately prior to execution. Although the compilation takes extra time, overall performance is improved because the slow interpretation is replaced by a direct and much faster execution of native code. JIT compilation achieves fast execution without sacrificing the portability of Java bytecode, and becomes an integral component of many Java virtual machines currently available. There are many successful JIT compilers which improve Java run-time performance using various methods such as efficient communication, I/O, register allocation,
method inlining, dead code elimination, and fast and efficient code generation.

AJIT\cite{16,56} uses an annotation system to collect useful program information during the JIT compilation process. The annotation conveys program information from the front-end translation phase to the code optimisation phase. The compiler generates annotations during the translation phase, and then goes through a mapping phase in which the bytecode operations are paired with the annotations and operations in intermediate representation, in order to generate the native code.

IBM presents an optimizing JIT compiler\cite{58,118,59} which reduces various run-time overheads via exception check elimination, common subexpression elimination, simple type inclusion test, method inlining, resolution of recursive method calls and others. In order to achieve cross-platform optimisation, it first uses a compact stack-based intermediate representation shared among all platforms, and then uses two register-based intermediate representations for advanced optimisations in the later phases, before architecture-specific optimisations such as register allocation and instruction scheduling being carried out. Based on the results of selectively applying different optimisations, \cite{59} identifies a set of most cost-effective optimisations which can achieve most of the overall performance improvement within a short compilation time.

Intel’s Java JIT compiler\cite{1} uses a lazy code selection approach for quick generation of good quality IA32 code. Instead of using any explicit intermediate representation, this approach directly uses bytecode to represent expressions and additional structures used on-the-fly. Therefore, it generates native code directly from bytecode in a single pass. Light-weight optimisations such as common subexpression elimination, register allocation, and elimination of array boundary checking are used.

Jaguar\cite{124} focuses on improving performance via efficient communication and I/O. It encapsulates system resources in Java objects (bytecode), and translates these bytecodes into short, inlined machine code segments which run much faster. In this way, Jaguar achieves high performance via direct native code access to system resources while retaining Java’s portability, type-safety abstraction and protection at bytecode level.

Kaffe\cite{127} uses a layer of macros written in native code, so that it can translate Java bytecode into native code in an "on demand" manner. It detects basic blocks and performs a single-pass code generation with fast register allocation.

LaTTe\cite{131} translates bytecode into pseudo code with symbolic registers, where many copies corresponding to pushes and pops between local variables and
the stack are then generated. It adopts a linear register allocation algorithm with local lookahead. However, its tradeoff between the profit and cost of this register allocation algorithm is carefully made for Sun SPARC, and therefore not generally applicable for other platforms.

Other work on JIT compilation includes Brikì[35][36], CACAO[54][75], OpenJIT[84], shuJIT[97], and Inprise’s JIT compiler in JBuilder for Linux[57].

The problem with JIT compilation is that there always exists an upper bound on the time a JIT compiler can use, which is the difference between the program’s original execution time (without JIT compilation) and the best execution time it can achieve via JIT optimisation. Only when a JIT compiler can finish all the analyses and transformations within this time bound can it improve the program’s runtime performance. This time bound is usually too short for the JIT compiler to carry out all comprehensive and in-depth analyses and transformations. Therefore JIT compilers normally use only light-weight optimisation techniques. Parallelisation is not feasible due to the time-consuming analyses it needs. Furthermore, the extra time needed for JIT compilation makes this approach less appealing when a quick response is a high priority requirement.

2.3.3 JVM optimisation

By specification[78], the Java virtual machine(JVM) defines an abstract stack-based computing machine on which Java bytecode runs. No specific hardware, operating system, or any particular implementation technology is assumed. It is demonstrated that in addition to optimising compilation technology, Java performance improvement can also be achieved via an efficient JVM implementation, which takes full advantage of the underlying hardware and operating system.

The Java HotSpot Virtual Machine[119][85] concentrates on several key areas in order to achieve high performance. These areas include on-the-fly adaptive optimisation, method inlining, redesigned and improved object layout, high performance memory system with fast and fully accurate garbage collection, use of native threads and fast thread synchronisation. Experimental results show that it can achieve at least a two-fold increase in speed for server-side Java applications.

Jikes Research Virtual Machine (RVM)[5][62], formerly known as Jalapeno[4], is a Java virtual machine which supports efficient and scalable execution of Java applications on SMP server machines. It employs a compile-only strategy. Jikes includes both a baseline compiler which translates bytecode directly into native code in the same manner as a Java interpreter, and an adaptive optimising compiler which provides three levels of optimisation. The latter uses a register-based
intermediate representation in order to enable more effective and machine-specific optimisations and bring greater flexibility in code motion and transformation. Experimental results show that it is capable of delivering performance comparable to the performance delivered by a production-strength JIT compiler. However, this achievement comes at the cost of Java’s portability.

The adaptive optimisation techniques used in both Java HotSpot Performance Engine and Jikes are introduced in section 2.4.2.

Other Java virtual machines, such as cJVM[9], Jupiter[42], Kaffemik[6] and JESSICA2[136][137][138], run multi-threaded Java applications in a distributed environment under a single system image.

2.3.4 Array optimisation

Scientific and engineering computation is characterised by intensive numerical computing which normally involves a large number of array references. Multi-dimensional arrays are not directly supported by Java [88], they are represented as an array of array(s) in order to provide general array structures as well as information needed for boundary checking. This makes access to multi-dimensional arrays expensive. In order to adapt Java to the area of scientific and engineering computation, NINJA[88][89][90][91][93] provides a package supporting true multi-dimensional arrays. A prototype compiler[14][15] is also developed to create safe and exception-free regions of code that can be aggressively optimised. It uses alias versioning and other transformations to automatically perform high order loop transformations and parallelisation. Experimental results show that it can deliver near-Fortran performance on numeric-intensive applications without sacrificing behavior in the presence of exceptions.

The drawback of NINJA is its lack of flexibility. First, it requires the array reference to be written in a different form from that which the language[53] specifies. Secondly, n-dimensional array (for n>3) is not directly supported. Furthermore, only arrays of basic numerical types (int, float and complex, etc.) are provided, and no direct support is provided to either create arrays of other existing classes, or to combine them in an easy and free manner.

2.4 Adaptive Optimization

2.4.1 Introduction

In traditional compilation, the optimisation is performed before execution. Once the executable is generated, no optimisation is possible that can utilise the run-
time context. The lack of knowledge about the target machine and input data set forces the compiler to make conservative assumptions to preserve correctness, and leads to a lack of flexibility and self-adaptability in different environment.

In order to cope with such difficulties, adaptive and dynamic compilers perform optimisations at runtime when knowledge of the target machine and data input becomes available. Adaptive optimisation [77] promises the elimination of interpretive overhead, common-case optimisation on demand, value-specific optimisation and control flow optimisation. It is classified into three major categories [121]: those that choose from statically generated code variants; those that modify behavior through parameterisation; and those that use dynamic compilation. A general description of execution-driven optimisation can be found in [39]. In [115], the state of the art in this area is reviewed, and a good description is given of the challenges inhibiting further acceptance of this technique. A survey on the evolution and state-of-the-art of adaptive optimisation in virtual machines is presented in [13] .

Adaptive optimisation needs dynamic code regeneration or composition to either directly replace old code segments of the executable with newly-generated ones, or to generate new code in source form that needs to be re-compiled before changes take effect. This enables the compiler to deliver a well performing executable by optimising the code dynamically, as the usefulness and correctness of optimisation techniques can be evaluated in the given execution environment.

Adaptive optimisation is different from JIT compilation in that the latter is mainly used for Java, and it compiles the program statically. No more optimisation is possible once the native code is generated, and the code remains unchanged during the execution. No runtime information is available for JIT compilation. On the other hand, adaptive optimisation is a general approach that can be used for any language (as demonstrated below), and the program is compiled and executed in a dynamic manner. The code may be modified during its execution to utilise the runtime information available.

In order to implement adaptive optimisation, a series of decisions must be made, such as what profile should be used, how to analyse profiles in order to identify particular candidates for code optimisation, what algorithm should be used for cost-benefit analysis, what optimisation techniques to be used, what should be the granularity of optimisation, in which order these optimisations should be applied and many others. Tradeoffs must be made regarding overheads, ease of use, effectiveness and other factors [77].
2.4.2 Adaptive optimisation for Java

2.4.2.1 Java HotSpot virtual machine

Adaptive compilation is also used in Java HotSpot virtual machine [119] in pursuit of high performance. HotSpot VM starts program execution using an interpreter without profiling, in order not to degrade the performance of short-running applications. The profiling does not start until the execution reaches a later stage. This approach helps to achieve profile of higher quality because a program’s behaviour may change during its execution. Execution hotspots are identified and optimised using techniques such as method inlining.

2.4.2.2 Jikes

Jikes RVM[5][62] adopts a dynamic optimising compiler[10][11][33] which is able to generate the best possible code of the selected methods for a given compile-time budget. Instead of using a stack-based intermediate representation, it uses a register-based one which not only enables more effective machine-specific optimisation but also brings greater flexibility in code motion and transformation. The program execution is monitored by Jikes’ profiler, and the profile is passed to Jikes controller[11]. The controller then makes an optimisation plan which is followed by the compiler to carry out efficient and aggressive optimisations at their respective intermediate representation levels. Experimental results show that Jikes’ performance is within an order of magnitude of the best current commercial compiler technology. A comparison between Jikes RVM and Java Hotspot VM is presented in [5].

2.4.2.3 Online feedback-directed optimisation of Java

In order to overcome the drawback of off-line profiling in feedback-directed optimisation (FDO), [12] implements a fully automatic online FDO system. Similar to Java HotSpot VM, it starts the program execution without instrumentation and profiling is not introduced until a later stage. Methods are profiled in depth only if found hot during execution. Besides timer-based sampling, it also uses a full-duplication instrumentation sampling framework in order to transparently reduce the overhead of executing instrumented code, in terms of impact on accuracy and code size. For each hot method, if the estimated benefit of optimisation outweighs the estimated cost, it is optimised in an online manner using four transformations: code splitting, code motion, method inlining and loop unrolling. Experimental results show that such an approach can substantially improve peak performance

24
whilst minimising the overhead.

### 2.4.2.4 Quasi-static compilation

Quasi-static compilation\[111\] introduced by the Quicksilver Java compiler is a new compilation approach that combines the benefits of static and dynamic compilation, without sacrificing compliance with the Java standard, in support of its dynamic features. It uses either a static or dynamic compiler to generate optimised binaries, and stores them in a reusable format called a *quasi-static image*. These images persist across multiple JVM instances and allow methods to be executed from the pre-compiled classes. The runtime compiler validates these images and adapts them to each specific JVM instance. If found obsolete, the images are regenerated at runtime. In this way the image may evolve over time to adapt itself to the dynamic environment. Quasi-static compilation allows the use of pre-compiled optimised code within a dynamic compiler whilst maintaining Java’s strict semantics. The Quicksilver compiler focuses on the integration of pre-compiled code within the adaptive compiler.

Quasi-static compilation is useful in scenarios where class files are invariant over executions of Java bytecode in distinct JVM instances, and where requirements on response time, performance, testability, reliability and memory are important. Experimental results show that it provides the benefit of the highest optimisation level whilst incurring compilation costs comparable to those of the fastest, non-optimising, dynamic compiler.

### 2.4.3 Non-Java adaptive optimisation

ADAPT\[121\][122][123\] is a compiler-supported high-level adaptive optimisation system. The user uses a domain-specific language to explicitly specify the optimisations available, as well as heuristics as to how to apply them dynamically at runtime. The ADAPT compiler reads these directives and generates an application-specific runtime system to apply the heuristics. The runtime system monitors the execution and the environment, periodically reviews the performance, and dynamically chooses the best code variants accordingly. ADAPT supports various adaptive optimisation paradigms such as dynamic compilation, parameterisation and runtime sampling. In ADAPT, optimisation is decoupled from the critical path of execution so that the code generation occurs concurrently with the execution. The runtime overhead is therefore minimised. This decoupling approach makes ADAPT unique amongst adaptive optimisation projects. Experimental results show that ADAPT consistently outperforms static optimisation alterna-
tives. Its main weakness is that its adaptive optimisation is not fully automatic, but relies on the user to provide the directives.

Similar to ADAPT, Dynamo[77] needs the user to specify in directives which optimisations to apply and when. It is a staged and selective dynamic compiler which can perform light-weight to heavy-weight dynamic optimisations according to user requirements. Different intermediate representations are used at different compilation stages in order to fulfill the corresponding optimisation tasks. Similar to ADAPT, the main drawback of Dynamo comes from its need for the user to provide optimisation directives.

Adaptive optimising compilers have to decide at runtime the right optimisation policy from all possible candidates. To avoid the complexity of decision-making, Dynamic Feedback[41] reduces all these policies to just a small, fixed number of options statically set in the system. At compilation it simply generates different versions of the same code segment using one policy for each. During execution, the generated code switches between sampling and production phases periodically. At the sampling phase, the overhead of each version is measured, and the feedback is used to choose the least-overhead version which will be used at the following production phase, until the next sampling phase comes. Dynamic Feedback has three main drawbacks. First, it tries only a small, fixed number of optimisation policies rather than all possible options. In addition, the cost it pays for phase switching and storing multiple code versions may affect the quality of profile. Finally, the phase switching interval is set fixed in the system, which may also affect the quality of profile. For example, if it is too short, the profile collected during one sampling phase may not be enough to accurately reveal the program's runtime behaviour, whilst if it is too long, the profile may lose its timeliness. As different programs have different runtime behavior, flexibility on this issue may help to achieve and maintain a reasonable accuracy and timeliness.

An infrastructure[30] is developed to integrate dynamic analysis and optimisation into a compiler. It considers only linear code streams and presents them in various presentations, each providing a different level of details whilst maintaining transparency with respect to the applications. An API is provided to the compiler writer to implement adaptive optimisations.

Quality of profile could be improved with help from the underlying operating system, or even hardware. Morph[134], a SUIF-based framework, combines operating system support and compiler technology in pursuit of continuous improvement in application performance. The underlying operating system helps collect timely, low-cost profile by statistical sampling. The Morph manager peri-
odically reads and processes the raw profile in an off-line manner, and the results are used to convert program modules, written in low SUIF intermediate representation, into native code. This reliance on environmental support for profiling is the main drawback of this approach. In addition, machine-specific information is needed during the optimisation. Morph’s need for operating system support makes it difficult to be used in a heterogeneous computing environment.

’C[43] is an extension of standard C which allows flexible, high-level, efficient and machine-independent specification of dynamic code generation. In order to reduce the size of the generated code, tcc[103], the ’C compiler, uses fast register allocation, efficient creation and composition of dynamic code specification and link-time analysis. Two different strategies are adopted to address the tradeoff between the code quality and the speed of dynamic code generation. The static back-end of tcc compiles the non-dynamic parts of ’C program and generates either native or C source for further optimisation, whilst its dynamic back-end is responsible for dynamic optimisation. Its main disadvantage is its limited adaptability because it simplifies the adaptive optimisation decision-making to just two different strategies.

Deco[45] is an automatic and selective dynamic optimisation system. Hot regions of program execution are identified by the Deco runtime profiler via path profiling, and then optimised by the Deco optimiser. The optimised code is put into a cos (code optimisation space) before the first instruction in the hot path of the original code is overwritten with a branch to the cos.

2.4.4 Summary

All the above projects demonstrate that adaptive optimisation can improve program performance by making use of information only available at runtime. However, there are still some drawbacks. First, at runtime, the time and resource spent on adaptive optimisation may affect both the execution and the quality of profile. For example, in the case of the multiple code variants approach, extra memory is needed to keep all the code variants and additional time is spent on scheduling. As these factors may affect performance, the number of different code variants must be kept low. This leads to the fact that it considers only a limited number of variants whilst the optimisation space may be large. In addition, in order to minimise this impact, light-weight optimisation techniques are widely used in most cases due to their low cost. High-level optimisations are capable of bringing performance improvement, as shown in [80], but the high cost of their implementation may offset the performance gain significantly. This is particularly
true if the adaptive optimising compiler does not have a distributed architecture which decouples the optimisation from the critical path of execution. This "light-weight optimisation only" approach leaves some performance improvement opportunities unexplored. Furthermore, in many adaptive optimisation projects, the tradeoff between the cost and performance gain is made statically. This lack of flexibility prevents adaptive optimisation from exploiting its full potential, as discussed in [41]. Finally, adaptive optimisation considers each execution independently. There is no way that the useful information obtained in one execution can be retained in order to benefit the future executions. Therefore, each execution must start its adaptive optimisation from scratch and can not make use of any prior knowledge obtained from previous efforts.

2.5 Iterative Optimisation

2.5.1 Concepts and model

The lifetime of a program is defined by [76][115] as the time between its first compilation and its last execution (inclusive). It is argued [115] that feedback-directed optimisation may be performed at any time during a program’s lifetime in order to keep adapting the program to its runtime context, which may change during its lifetime. In iterative compilation[70][71], successive transformations are applied to the program, their worth determined by either actual execution of the resulting code, or by estimation. A large number of different versions of the program are generated and evaluated, from which the best version is selected. This approach does not suffer from compile-time undecidability and, given sufficient time, can find the best program at the cost of increased time of successive compilation.

During the iterative optimisation process, the compiler repeatedly executes different program versions, collects and stores the profile, analyses the profile, decides the next optimisation attempt, generates the new version of program for the next attempt and compiles it, as long as the budget allows.

Iterative optimisation is based on searching in an optimisation/transformation space. As [70][102] demonstrates, the optimisation space is highly irregular in shape and highly non-linear with many local minima. Preliminary experimental results show that iterative compilation is a feasible and effective approach, capable of finding good optimisations by visiting a relatively small fraction of the entire optimisation space. It is now becoming increasingly viable for general-purpose computing.
Iterative optimisation is similar to adaptive optimisation in many aspects. First, both use runtime information to adapt the program execution to the runtime environment. Furthermore, both need to make decisions amongst all the available options. Finally, both use runtime feedback to make these decisions.

However, there are many differences between these two approaches. First, adaptive optimisation is considered a one-off approach, because all of its effort is restricted to one execution, and no optimisation is possible afterwards. On the contrary, iterative optimisation allows optimisations to be carried out throughout many executions within the program's life time. Secondly, because of its one-off manner, adaptive optimisation must include within the only execution all the variants it could consider, usually a small number. However, iterative optimisation usually considers only one option/variant at a time. Budget allowing, it can consider, within different executions, as many variants as possible. The variety it considers is therefore much larger than that considered by adaptive optimisation. Thirdly, the multiple execution manner allows iterative optimisation to consider high-level optimisations which are infeasible for adaptive optimisation due to the cost (as explained above). This is because the impact of their implementation on program performance could be minimised, if the transformations are carried out during the "idle time" [76], i.e. between the executions. Furthermore, iterative optimisation usually considers only performance and uses it to direct the search in the potential optimisation space, whilst adaptive optimisation usually considers more runtime information (runtime constants, for example). Additionally, in the case of adaptive optimisation, profile information becomes useless after execution and is therefore discarded. However, it is stored for future use in the case of iterative optimisation. Finally, adaptive optimisation must be carried out in an online manner, in which the collected runtime information is used instantly for optimisation, whilst iterative optimisation is usually carried out in an offline manner, as explained below.

Iterative optimisation could be carried out in either online or offline manner. Online iterative optimisation means that the optimisation is carried out in parallel to the execution and profiling, so that the new version does not necessarily have to wait until the next execution to be brought into play. On the contrary, in offline iterative optimisation, each optimisation attempt is a separate execution of the program, only after it is finished will the compiler make the decision for the next attempt and carry out the corresponding optimisations at idle time, and only after the compiler finishes all the work will the next attempt start.

Because the cost of memory and CPU time spent on the iterative optimisation
is unavoidable and may be high, it may seriously affect the quality of profile and consequently the quality of optimisation. In order to minimise this interference, iterative optimisation is normally carried out in an offline rather than online manner. Offline iterative optimisation may be used in areas such as embedded systems and scientific and engineering computing, where the long-term benefits can offset the cost of long optimisation time.

However, this does not mean that online iterative optimisation is not feasible. If a compiler has a distributed architecture similar to ADAPT, the optimisation can be decoupled from the critical path of execution by having the client end profile the execution at a small cost and the server end perform optimisation. On the other hand, current dynamic optimisation techniques are capable of replacing code segments with optimised versions even during execution, as Deco[45] shows. These two factors make online iterative optimisation feasible, although there is no published work using this approach.

### 2.5.2 Search in parameterised optimisation spaces

There have been some successful projects in the area of iterative optimisation. They mainly focus on a simplified and regular optimisation space composed of a certain number of parameterised transformations (mostly in fixed phase order). Different approaches are developed to explore such a parameterised optimisation space. They are briefly reviewed in this subsection.

#### 2.5.2.1 Grid-based search

A grid-based search algorithm is presented in [70][71]. It focuses on a regular \( n \)-dimensional transformation space composed of only a small number of parameterised transformations. Each of these transformations is considered as an axis in the space, with each point on the axis representing a value of the transformation parameter. The search algorithm first defines a coarse grid on the space, evaluates all points on the grid, and orders them in a priority queue, before refining the grid around the head point of the queue. These above steps are repeated until an optimal point is found or the optimisation budget is reached. The space which [70][71] consider is highly regular and very small, composed of only two transformations (loop tiling and loop unrolling, each with a small range of parameters) in fixed phase order. Experimental results show that this algorithm can find optimal parameter combinations in a relatively small number of steps, and only a small fraction of the entire search space is visited.
This algorithm is also used in [80] to investigate the impact of high-level transformations on Java performance. The experimental results show that combining high-level transformations can improve Java’s performance significantly, and good transformation combination can be found relatively quickly.

Grid-based search is demonstrated to be very effective for search in a space of parameterised transformations. Its shortcoming is that the search space it considers is not only low in dimension but also highly regular, therefore its efficiency remains a doubt when it comes to a less regular space with a large number of transformations, although the space can be casted to a much larger and regular one with high dimension. Furthermore, its effectiveness is yet to be confirmed on benchmarks of larger size than the kernels.

2.5.2.2 Random search and gradient search

Besides grid-based search, random search and gradient search[70] are also examined on the same space as described above. Experimental results show that both of them are consistently outperformed by grid-based search.

2.5.2.3 Tree/grid-based search

A larger optimisation space than that in [70][71] is considered by [102]. It contains more transformations which are arranged in different but fixed orders. Therefore the space is larger, more complicated and less regular. Trees are used to represent non-parameterised transformations, and grids are used to represent parameterised transformations. Accordingly, a tree-based search is used to look at the transformation space, except in the case of parameterised transformations where the grid-based search is used. The combined tree/grid space is traversed and the best optimisations are added to the priority queue.

Two strategies[102] are developed to explore this optimisation space. One is transformation-based and the other is cost-based. Three architecture-blind generic optimisation approaches are implemented and tested, but backtracking is not involved. Experimental results show that iterative compilation is a viable approach to program optimisation in cases when high performance is crucial and long compilation run is affordable.

2.5.2.4 Phasewise exhaustive search and random search

Two different random approaches are used in [50] to search in an optimisation space composed of three parameterised transformations in array padding, loop tiling and loop unrolling. One is a phasewise exhaustive search algorithm that
carry out the search in a fixed phase order manner, in order to avoid combinatorial explosion. It first considers array padding where all pad sizes are tested and the best is chosen for the next step. Loop tiling is then applied in the same manner, followed by loop unrolling. The other approach just randomly selects one or more transformations from the three, and then applies them with randomly chosen parameters. Experimental results on three full SPEC FP benchmarks and six different platforms show that both approaches can significantly outperform a naive compiler with full optimisation.

Although the phase-wise exhaustive search algorithm significantly avoids combinatorial explosion, it is based on the following assumption yet to be confirmed: that any sub-sequence starting from the head node of the optimal transformation sequence is optimal concerning the corresponding transformation space. This is the main drawback of this algorithm.

2.5.2.5 Search with heuristic pruning and static cost model

A compiler is introduced in [128] that searches for the optimal optimisation in a restricted optimisation space which is based on a fixed order of five loop reordering transformations (fusion, fission, unrolling, tiling and interchange). Instead of using runtime feedback for evaluation, it adopts a static cost model and uses an aggressive heuristic pruning algorithm in order to control the complexity of the search in the entire optimisation space. Experimental results demonstrate good efficiency in the resulting code and a short running time for the search. However, it is difficult to generalise this approach because the performance estimation is based on the in-depth understanding of the interactions among these five optimisations, which is inevitably architecture-specific. This approach can be enhanced by the framework presented in[135], which integrates code models, optimisation models and cache models to predict the impact of applying some loop transformations for data locality.

2.5.2.6 Compiler Optimization-Space Exploration

Predictive heuristics[17][55] are useful in indicating where and how the code transformations should be applied in the search for performance improvement. They are widely used in modern optimising compilers. However, precision of these heuristics is difficult to achieve in practice, due to their complexity and the interference among them. Decisions are usually made conservatively, which results in suboptimal achievement and potential performance improvement opportunities left unexplored.
An optimization-space exploration (OSE) compiler organisation technique is presented in [120]. It considers the optimisation space as derived from a list of parameters, each of which controls either the application of a transformation, or the optimisation levels. Each point in the space is therefore an optimisation configuration, which is equivalent to a transformation sequence. Based on the observation that performance improvement is usually decided by a few critical optimisations, in compiler construction time, OSE uses the compiler writer’s prior knowledge to remove configurations that are not likely to improve performance. At compile time, it evaluates a small set of the remaining correlated configurations, before using the results to select more for further evaluation. OSE uses a compile-time performance estimator instead of runtime profile. This enables the OSE compiler to apply multiple configurations to the code and evaluate their achievements simultaneously.

OSE approach has several drawbacks. First, the compiler writer’s experience is inevitably architecture-specific, and might not apply in a new architecture. When OSE uses this experience to prune the optimisation space at the first step of compiler construction, it may leave many areas unexplored, where optimisation opportunities may exist. In addition, the performance estimator is also architecture-specific. Considering the difference of various architectures in resource utilisation, cache usage and other factors, it is difficult to develop high quality estimators generally applicable. OSE has poor portability because of the above two factors. Furthermore, it considers a small number of parameters, each with only a small set of valid values. The resulting optimisation space is relatively small.

2.5.3 Search in non-parameterised optimisation spaces

It is obvious that in practice, the optimisation space that the compiler must consider is more complex than the highly regular ones discussed in the above subsection, due to the large number of existing transformations. There has been some work attempting to tackle this problem, as discussed below.

2.5.3.1 Biased random approach

A prototype compiler framework is presented in [38] that uses a biased random search algorithm to discover a program-specific transformation sequence that minimises an explicit, external objective function. It can adapt itself to the runtime environment in which it operates, to the application it compiles, to the transformations it uses, to the objective function, and to the target architecture.
This framework considers all the available transformations in a pool and constructs a transformation sequence by selecting transformations from the pool and deciding the parameters arbitrarily if needed. These transformations are applied to the program in a global manner and in the order of their appearance in the sequence. The legality is guaranteed by the data-flow nature of the transformations considered. An adaptive random sampling algorithm is used to steer the search. A probabilistic model is built and used to bias the search and explore the optimisation space in depth. Preliminary experimental results show that in each case the adaptive compiler outperforms the original, fixed-sequence compiler by providing shorter sequences in most cases.

This approach tries to solve the phase order problem\cite{72} a compiler faces, which is relatively easy if compared to the problems faced by other projects summarised in this chapter. It has several drawbacks. First, it cannot accumulate and keep any knowledge from its previous attempts on how to avoid transformation interference, which is inevitable in many transformation sequences. Secondly, the global manner of transformation application is naive. For example, the benefit of properly applying one transformation to one code segment may be offset by improperly applying the same transformation to another code segment. Efficiency may be improved by making the transformation application hotspot-aware. Finally, parameters could be chosen in a smarter way, such as by using a grid-search or profile from previous successful attempts.

2.5.3.2 Problem-specific iterative optimisation

Systems for generating highly-optimised BLAS routines are presented in \cite{125}. These systems probe the underlying hardware to find optimal values for tile sizes, unrolling factors, etc, but they are not general-purposed compilers. Experimental results show that they are capable of producing efficient codes.

A method is introduced in \cite{27} to search for the best optimisation on the assembly level of embedded applications, taking into consideration both execution time and code size. It uses a static cost model and does not include a pruning algorithm.

2.5.4 Lifelong program analysis and transformation

Although the optimisation approaches currently available are effective, they usually focus on one particular phase of a program’s lifetime. For instance, adaptive optimisation focuses on optimisation at execution time, when runtime information becomes available. In \cite{76}, it is believed that the program analysis and
transformation shall be carried out at all stages of a program’s life time, rather than be restricted to any particular one. It highlights the demand for a unified representation of program information so that different optimisation techniques can share useful information and keep optimising the program during its life time.

The Low-Level Virtual Machine (LLVM)\textsuperscript{[76]} is a compiler framework aiming to support transparent lifelong program analysis and transformation. Extensive program information (such as type information, control flow graph and dataflow information) is represented in LLVM using a unified, low-level and typed code representation. Once collected, this information is preserved throughout the program’s life time, from analysis, transformation to distribution. It enables LLVM to provide persistent program information, offline code generation, user-based profiling and optimisation, transparent runtime model and uniform whole program compilation. Optimisations can therefore be carried out at compile time, link time, install time and run time as well as at idle time (between runs). This idea of unified, low-level and persistent program representation is very similar to the quasi-static image introduced in \textsuperscript{[111]}. However, quasi-static compilation does not have all these five capabilities mentioned above.

Furthermore, LLVM is source-language-independent, which means that it can accept programs written in arbitrary language. However, it does not provide the "compile once, run anywhere" architecture neutrality, as it still needs a native code generator for compilation.

\section{Systematic Exploration of Optimisation Space}

\subsection{Introduction}

The Unified Transformation Framework (UTF) \textsuperscript{[65]} aims to provide a uniform way to represent and reason about transformations, based on the idea that an iteration reordering transformation can be represented as a mapping from the original iteration space to a new iteration space. It unifies iteration reordering transformations that can be obtained by arbitrary combinations of loop interchange, loop distribution, loop reversal, loop fusion, loop distribution, loop alignment, loop interleaving, loop scaling, loop coalescing, loop skewing, loop tiling, index set splitting and statement reordering. A detailed review of UTF will be given in the following chapter.

UTF is a generalisation of the unimodular transformation\textsuperscript{[18]}. It provides a mechanism to represent dependencies, to test the legality of mappings\textsuperscript{[68]}, and to align mappings. A code generation algorithm\textsuperscript{[66]} is also provided which can
generate code from the mapping(s) given. In addition, techniques[68] are provided
to infer constraints on the legal mappings for individual statements or on the
relations between mappings for different statements, and to infer constraints on
the legal mappings for some statements, based on the mappings given for other
statements. All these techniques help to construct legal mappings in an easier
way.

It is worth noting that UTF itself is not capable of deciding which transfor-
mation to apply. What it provides is a simple setting to solve this problem. The
transformation selection problem is left to the surrounding system such as [67].

2.6.2 Mapping construction based on performance esti-
mation

A UTF-based algorithm is presented in [67] to decide which transformation(s)
should be applied to a given program. This algorithm is closely related to the
A* heuristic search algorithm[98]. It constructs the mapping for each statement
in a level by level manner. At each level, an estimation is made on the partly
specified mapping. The latter is augmented if and only if its performance estimate
is "sufficiently good" compared to the the performance of the current tentatively
accepted mappings. This level by level extension finally makes the mappings 1-
1. Backtracking is used during the construction process, which ceases to extend
those tentatively accepted mappings no longer "sufficiently good" and removes
them from the candidate list.

Three different performance estimators are used in [67]. They are simplicity,
locality and parallelism granularity. This paper also provides a mechanism to
trade-off these independent program properties. Experimental results show that
this algorithm is capable of obtaining good results in a feasible amount of time
for kernel-sized programs.

The drawback of this algorithm is that there is no run-time feedback to be
used to bias the mapping construction. Since no code generation is possible from
a partly finished mapping, a static cost model is used instead to estimate the
performance as run-time feedback. So, the quality of the performance estimator
is crucial but difficult to achieve on non-trivial cases. In addition, the efficiency
is yet to be confirmed on large benchmarks.

2.6.3 Genetic Algorithm Parallelism System

The Genetic Algorithm Parallelism System (GAPS) [99][100] is a UTF-based com-
piler framework which uses a novel iterative feedback-directed approach to auto-
matic parallelisation. This approach is based on the following ideas about genetic algorithm optimisation: an encoding describes a legal ordered set of transformations; mutation and recombination reproduction operators describe mechanisms that alter encodings; and an evaluation function represents a target architecture plus the program to be compiled. It calculates the overhead of the program when the latter is transformed, giving high fitness values to encodings which yield low overheads.

GAPS aims to evaluate the application and performance benefit of genetic algorithm optimisation techniques to the compilation of loop-based programs for parallel architectures. Six reproduction operators are used here, which are group-based mutation, group-based crossover, iteration-based crossover, padding mutation, tile mutation and unroll mutation respectively.

For legal encodings/mappings, GAPS generates, compiles and runs SPMD code. It measures the time taken to execute SPMD code on two processors, and the results are linearly normalised to provide a fitness which determines the individual's selection probability. Illegal encodings/mappings are labelled "bad", and legal ones always have a greater probability to be chosen for reproduction than illegal ones.

Preliminary experimental results show that GAPS can produce encouraging performance improvements, but this achievement takes a very long time, due to the random nature of genetic algorithm. GAPS generates a very large number of mappings, many of which are illegal. But it has neither a bias mechanism for reproduction operator selection nor a mechanism capable of either remembering good mappings previously generated, or accumulating knowledge from previous attempts in order to avoid generation of illegal mappings.

Project Iterative Compilation Environment (ICE)[101] utilises GAPS techniques to investigate the architectural independence of genetic algorithm techniques. It allows a wider range of transformations to be applied by the genetic algorithm, including array padding, loop permutation, loop fission, loop fusion, loop unrolling and loop tiling. In addition, profiles are used in order to give high selection probability to the statements accounting for large portions of a program's execution time.
2.7 Optimisations Using Machine-Learning Approaches

2.7.1 Introduction

As discussed in the previous sections, an optimising compiler’s task of finding the optimal transformation combination or sequence can be thought of as searching for a corresponding optimal point in an optimisation space. It is understandable that the knowledge a compiler learns from its previous attempts will be helpful in its new search, even if it is asked to optimise another program in a different environment. Machine learning[94] is concerned with the question of how to construct programs that automatically improve using prior knowledge. Therefore, it looks promising to introduce machine learning techniques to optimising compilers, so that they can evolve and adapt themselves to changes in the software and hardware environment.

A learning compiler can be thought of as an iterative optimising compiler with machine learning enhancements. It successively applies different transformations to the program, evaluates their performance, and collects the profile information during the course, as an iterative optimising compiler does. However, it is different in that it uses machine learning techniques to learn prior knowledge not only from the profile of the current program, but also from the profile of previous programs it has met. This prior knowledge is retained by the compiler and later will be applied to guide further exploration of the given program, or to compile another program, or even to compile a program in another environment, as suggested before.

A list of tasks are summarised in [94] about the development of a machine learning system. Key issues include the target function, representation of the target function, the function approximation algorithm and training example selection. In the optimising compiler case, target function stands for the optimisation target(s) such as execution time, code size, power consumption, etc. or any arbitrary combination of them. Target function representation concerns about how to represent the target as a function of both the transformations applied and the characteristics of the program and runtime environment. The function approximation algorithm is about how the compiler learns from its previous attempts. Training example selection considers how to choose training cases for the compiler to learn efficiently and impartially. All these issues must be addressed in order to develop a learning compiler.

The main target of a learning compiler could be summarised as to build up
a virtual model of both the software and hardware environment, which could help the compiler to decide, without execution, what is the best optimisation for a given program. This virtual model could be represented in various ways, depending on the optimisation target, the features of the environment, the learning approach chosen and many other factors. The training examples would be obtained from various code variants (of different programs and different optimisations) and their corresponding performance in the target environment.

In recent years, there has been important and successful progress in theory, algorithms and application of machine learning. It has been demonstrated of great practical value in a variety of application domains. Some of these projects successfully use machine learning approaches in optimising compiler development, as summarised below.

2.7.2 Cased-based reasoning for interactive performance tuning

In [86], an interactive tool is introduced to complement existing compilers and automatic parallelisers and to help users with the performance tuning process. It uses a case-based reasoning approach which tries to solve a given problem by adapting the solution of a similar case already encountered. In order to identify and retrieve cases, a set of code properties are used as indices of loop properties and statement patterns. A mechanism is provided to add new cases into its knowledge base. Experimental results show that even with just a limited number of cases, this approach works well to help users recognise optimisation opportunities.

2.7.3 Classification based on statistical group performance

Machine learning is used in [87] to address the problem of automatic generation of optimisation heuristics. It considers a small optimisation space consisting of just one transformation: loop unrolling. Five sets of integer features are used to describe the target loop. Training cases sharing common abstractions are grouped into classes. Each class is then labelled positive if loop unrolling improves the performance for the majority of the class, or negative otherwise. The unrolling decision rules can therefore be represented as a decision tree, each node of which corresponds to a certain class and represents a test checking the value(s) of one or several features. In this way the machine learning process is cast into building the decision tree. When a new loop arrives, it is classified as a leaf loop class in the tree, and then the corresponding unrolling heuristic is applied. Experimental
results show that such a decision tree that provides specific heuristics for loop unrolling can be learnt.

The drawback of this approach is its poor extendibility when more features are introduced to specify the loop. Heuristic tree pruning and restructuring algorithms are needed to maintain the efficiency of the decision tree. On the other hand, if the optimisation space is enlarged with more transformations, there may exist many heuristics that can bring performance improvement, as shown in this thesis. This approach had not addressed the problem of heuristic selection in such cases.

2.7.4 Genetic algorithm for DSP optimisation

It is claimed in [114] that many digital signal processing (DSP) algorithms can be represented as a transformation matrix which is multiplied by an input data vector to produce the desired output vector. The matrix can be factorised for faster implementation and represented by a very large number of formulae which are different in appearance but equivalent mathematically. These formulae may vary very significantly in runtime performance. This means that for each transformation matrix, there exists a large formula space and there is a wide variance in performance of these formulae. One of the problems in this area is to find in this space a formula as efficient as possible. Due to the size of the space and the complexity of modern processors, it is difficult to analytically predict the performance of these formulae, or model it by hand.

STEER[114] is a stochastic evolutionary algorithm capable of searching through this large space of possible formulae with a larger coverage. It is a variation of standard genetic algorithm in that it uses split trees rather than bit vectors as its representation. Experimental results on the Walsh Hadamard Transform (WHT) show that STEER can find optimal formulae whilst still profiling a very small portion of the space.

Furthermore, the search is considered by [113] as a control learning problem. With the help of reinforcement learning and an accurate cache miss predictor based on several DSP observations, it can effectively learn to construct fast WHT formulae across many sizes after only be trained on data from one particular size. Due to the nature of genetic algorithms, this achievement comes at the cost of checking a large number of formulae, but is considered affordable in the DSP domain.

This approach aims to solve a specific DSP problem, therefore, it is not generally applicable. There is no evidence about its applicability on other DSP
transforms and on other platforms.

2.7.5 Genetic programming for compiler heuristic optimisation

A compiler algorithm usually contains a number of options which an optimising compiler needs to prioritise in pursuit of high performance. It is a common feature that such priority functions are prevalent in compiler heuristics. Due to the nonlinearities within the compiler and the complexity in the processor architecture, it is difficult to hand-optimise these priority functions.

Such priority functions are considered as executable expressions in [116][117], which uses genetic programming (GP)[74] to optimise them. It considers the priority functions associated with three compiler heuristics: predicated hyperblock formation, register allocation and data prefetching. Experimental results show that GP can achieve, at the cost of long optimisation time, impressive results that outperform the existing heuristics.

This approach allows the compiler heuristic to evolve. The resulting heuristic is applied to all subsequent programs. However, it is not able to adapt itself to the runtime information of subsequent programs.

2.7.6 Machine learning for instruction scheduling

Instruction scheduling can be thought of as a learning task. For local instruction scheduling, which deals with a basic block of instructions, there is a greedy approach which iteratively selects the best instruction (as determined by some heuristics) from those available, proceeding from the beginning to the end of the block. By representing a partial schedule as a triple, this task of selection is thought by [95] to be like learning a preference relation over triples. Those triples belonging to the relation define pairwise preferences in which the first instruction is considered better than the second. The problem is thus framed as a supervised learning task in which the learning algorithm tries to construct a suitable predicate of the target concept, based on the positive and negative examples presented to it. An approach is developed to correctly label the examples and counterexamples needed for the learning task. A variety of learning algorithms have been tried in order to fulfill the task of updating the expression of the preference relation. They include decision tree induction program, table lookup, function approximator and feed-forward artificial neural network. Empirical results on the Digital Alpha platform show that they all produce a scheduler better than those
used by the production compilers, but not as good as the heuristic scheduler provided by the processor vendor.

List scheduling is a method often used for instruction scheduling. Its essence is to heuristically decide which data-ready node to schedule next. Many list scheduling approaches assign to each node a weighted sum of key scheduling parameters as its priority. It is suggested that these weights be experimentally determined. In [19], twenty-four parameters are considered in order to search the space of possible weights via a standard genetic algorithm, so that these weights can be tuned to the specific needs of a scheduler/architecture pair. Experimental results on three different architectures show that a genetic algorithm is able to tune the list scheduler to a particular architecture/scheduler pair. As the weights vary significantly not only on different architectures, but also when moving from local scheduling to global scheduling, this demonstrates that genetic algorithms can bring significant adaptability to the instruction scheduler.

Similar to STEER[113][114], the main drawback of these two learning approaches is that they both try to solve a very specific problem, therefore, they are not generally applicable for the optimisation problems a general-purpose optimising compiler faces, as specified above. In addition, both of them need a large set of training examples.

2.8 Summary

This chapter has given a brief review of areas of work related to that which is presented in this thesis.

It started with a brief review of Java technology, including the Java programming language and the Java virtual machine, then summarised various approaches developed to achieve performance improvement in Java. Following this, a summary of adaptive optimisation techniques has been given, and some successful works in this subject summarised. Iterative optimisation has been reviewed and different approaches discussed. Pugh’s Unified Transformation Framework (UTF) provides a simple setting to consider program optimisation as an exploration of the optimisation space. It has been briefly introduced before some UTF-based approaches were discussed.

Finally, a brief summary of machine learning has been given, including ways it could be used in optimisation. The concept of a learning compiler has been introduced and previous work in this area has been discussed.
Chapter 3

Search Space

3.1 Introduction

Program optimisation can be considered as searching in an optimisation space for the best point(s). This space is briefly introduced in this chapter, with some examples demonstrating its complexity. It is followed by a review of the Unified Transformation Framework (UTF), including its mapping notation and the facilities it provides. The search space this thesis considers is then explicitly specified with the help of a naive exhaustive scan algorithm, before an example is given to demonstrate the size of the space. A set of program features are used to describe this space. They also enable the compiler to correlate programs’ structures against performance in order to search for optimisation opportunities. These features are listed at the end of this chapter.

3.2 The Search Problem

Various approaches and techniques [2][26][96][104][129][130] have been developed and widely used by modern compilers to achieve performance improvement. Many of them have special legaility constraints and transformation rules. Their combination leads to a large optimisation space for the compiler to explore.

This optimisation space is not only large but also complex, due to the interference among the transformations and the program to be optimised. For example, Figure 3.1 shows the results of applying loop tiling to four code segments chosen from the Java Grande Forum Benchmark Suite[32]. For each code segment, all tile sizes from 1 to 100 were applied, and the performance improvement ranged from no improvement at all (speedup 1.00) up to 41% improvement. In the case of A and D, loop tiling finds no speedup. In the case of B and C, we see much greater improvement of 41% and 40% with a tile size of 14 and 12 respectively.

43
This is due to the relatively large data sizes accessed in both programs. Similarly, loop unrolling was applied to each of the innermost loops of these four code segments, with all unrolling factors from 1 to 20. The results are shown in Figure 3.2. It shows that loop unrolling does not bring much performance improvement to B and D, due to their relatively large loop bodies. For A, C and D, the best performance occurs with an unrolling factor of 2, 19 and 4, respectively, with the performance improvements reaching up to 23%. The results of applying various transformation combinations on these four code segments can be found in [80]. Results of similar experiments can also be found in [50] and [70][71].

These results demonstrate that even with just one or a few transformations, the resulting optimisation space is highly non-linear and contains many local minima as well as some discontinuities[70]. It is infeasible to analyse or predict the performance using static approaches. The combinatorial explosion of the optimisation space when more transformations are introduced adds more difficulties to the search problem.

Therefore it is interesting to determine how the compiler should best explore this optimisation space. Much work has been done in this area, as discussed in the previous chapter. Many of them focus on a very small subspace of the whole optimisation space, consisting of a small number of transformations. In
such circumstances, a random approach is feasible and achieves good results as demonstrated in some iterative optimisation projects. It is acknowledged that these good results may be optimal points in the subspace the compiler can explore. However, due to the complexity of the whole optimisation space, it is hard to decide whether or not there exist even better points outside these restricted subspaces, how good they are, and how many of them there are.

It is believed that a systematic approach, which can arrange the search space exploration in an organised manner, will be able to provide more convincing results concerning the coverage. Such an approach relies on the support of a unified representation of the space it is to explore. There have been some efforts aiming to provide such a kind of support, as reviewed in the Related Work of [65].

An ideal unified representation of the optimisation space would be able to represent all the transformations (available currently or in the future) and their legality constraints. But such a target is currently considered too ambitious, considering the diversity of the transformations.

This thesis uses the mapping notation presented in Unified Transformation Framework (UTF) [65] to specify a reasonably large optimisation space. Although not able to include all available transformations, it contains most of the iteration
reordering transformations UTF can represent. In addition, UTF provides many useful supporting facilities which help to specify a search space and organise its exploration.

3.3 Unified Transformation Framework (UTF)

The Unified Transformation Framework (UTF)\cite{65} aims to provide a uniform way to represent and reason about iteration reordering transformations. Although incapable of deciding what transformation(s) to apply, it provides a simple setting to solve this problem, with the algorithms for legality tests, mapping alignment and code generation.

3.3.1 Mapping

A statement’s iteration space is defined in \cite{69} as the set of iterations for which the statement will be executed. In the iteration space, all the points are executed in a lexicographic order. Certain reordering of these points could be represented by a unimodular transformation\cite{18}, which specifies any transformation obtained by composing loop interchange, skewing and reversal. Such a reordering can be considered as a mapping from the old iteration space to the new one. UTF generalises the idea of unimodular transformation in order to describe a larger class of such mappings.

In UTF, a mapping is expressed\cite{65} as:

\[
T : [i_0, ..., i_m] \rightarrow [f_0, ..., f_n] | C
\]  

(3.1)

where:

\(i_0, ..., i_m\): iteration variables

\(f_0, ..., f_n\): functions, usually quasi-affine functions

\(C\): optional restriction

It represents the fact that, if condition \(C\) is true, iteration \([i_0, ..., i_m]\) in the original iteration space is mapped to iteration \([f_0, ..., f_n]\) in the new iteration space.

The \(f_i\) expressions are called mapping components and the positions are referred as levels. Each mapping component is either a syntactic component (always an integer constant) or a loop component (a function of the loop variables for the statement). Loop and syntactic components can be grouped into loop vector and syntactic vector respectively.

46
for (i = 0; i<1024; i++)
    for (j = 0; j<2048; j++) {
        0: b[i][j] = c[i] + d[j];
        1: a[i][j] = c[j] + d[j];
    }

\[ T_0: [i, j] \rightarrow [0, i, 0, j, 0] \]
\[ T_1: [i, j] \rightarrow [0, i, 0, j, 1] \]

Figure 3.3: A simple loop example and its identity mappings

for (i = 0; i<1024; i++)
    for (j = 0; j<2048; j++) {
        1: a[i][j] = c[j] + d[j];
        0: b[i][j] = c[i] + d[j];
    }

\[ T_0: [i, j] \rightarrow [0, i, 0, j, 1] \]
\[ T_1: [i, j] \rightarrow [0, i, 0, j, 0] \]

Figure 3.4: The loop after applying the mapping which swaps the last syntactic components

Each loop component contains a constant part and a variable part which is a quasi-affine function of the loop variables. Quasi-affine functions are linear functions plus integer division and remainder when divided by a constant. These functions of a mapping are of primary importance in determining the parallelism and data locality of the code resulting from the mapping, which could be explored for performance improvement. Intuitively, the syntactic components represent the loop structure as well as the statements’ positions in the loop. For instance, consider the two statements in the double nested loop in Figure 3.3, the syntactic vectors of their identity mappings (which intuitively means no transformation is applied) are (0, 0, 0) and (0, 0, 1) respectively, as shown in \( T_0 \) and \( T_1 \). If they are (0, 0, 1) and (0, 0, 0), the transformed loop will be as shown in Figure 3.4. The loop structure will be as shown in Figure 3.5, if the syntactic vectors are transformed into (0, 0, 0) and (0, 1, 0) instead.

The dimensions of the loop and syntactic vectors are called loop depth and syntactic depth respectively. The relations among loop depth \( l \), syntactic depth \( s \) and mapping depth \( m \) are

\[ s = l + 1, m = s + l. \quad (3.2) \]
for (i = 0; i<1024; i++)
    for (j = 0; j<2048; j++) {
        0: b[i][j] = c[i] + d[j];
    }
    for (j = 0; j<2048; j++) {
        1: a[i][j] = c[j] + d[j];
    }

$T_0: \[i, j\] \rightarrow [0, i, 0, j, 0]$
$T_1: \[i, j\] \rightarrow [0, i, 1, j, 0]$

Figure 3.5: The loop after applying the mapping which hoists the last syntactic components to a higher level

For example, given a mapping $T : [i, j] \rightarrow [0, i, 0, i + j, 0]$, its loop vector is $(i, i + j)$ and its syntactic vector is $(0, 0, 0)$. Its loop depth and syntactic depth are 2 and 3 respectively, which result in the mapping depth of 5.

UTF is an extension of unimodular transformations[18]. It specifies a separate mapping for each statement in the target code segment. For convenience, we call any of these combinations a schedule where a mapping is assigned to each of the statements in the target code segment. The $f$s in a mapping are allowed to include a constant term, and they may include integer division and modular operations, provided the denominators are known integer constants. It is worth noting that the dimensionality of the old and new iteration spaces could be different. Furthermore, the mappings could be piecewise, i.e. a mapping could be a union of mappings with disjoint domains.

With the above extensions, UTF can represent a large set of iteration reordering transformations [65] that can be obtained by any combination of loop interchange, loop reversal, loop skewing, statement reordering, loop distribution, loop fusion, loop alignment, loop interleaving, loop tiling, index set splitting, loop coalescing and loop scaling. It provides a uniform way to represent and reason about iteration reordering transformations, although no dataflow optimisation is included.

3.3.2 Dependence

In UTF, dependences are represented and evaluated using linear constraints over integer variables, which appear in the form of tuple relations. The Omega test[106] is used to simplify and manipulate these constraints. It also enables accurate dependence composition and provides more useful information.
UTF expresses all the control dependences implicitly by using a set of linear
inequalities on the loop variables and symbolic constants to describe the itera-
tion space. This is achieved by requiring that conditionals be removed using
if-conversion[3] and all loop bounds be affine functions of the surrounding loop
variables and symbolic constants.

Data dependences are represented as tuple relations. For instance, the depen-
dence from \( s_p \) to \( s_q \), which is caused by a dependence from \( s_p[i] \) (i.e. iteration \( i \)
of statement \( s_p \)) to \( s_q[j] \), is represented by tuple relation \( d_{pq} \) which maps \([i]\) to \([j]\)
\((i, j \text{ are tuples})\).

### 3.3.3 Legality test

A mapping is *legal* if the transformation it describes preserves the semantics of
the original code. This is true if all of the dependences in the original code are
respected by the new ordering of the iteration space.

Therefore, the legality requirement is as follows[65]: if \( i \) is an iteration of
statement \( s_p \) and \( j \) an iteration of statement \( s_q \), and the dependence relation \( d_{pq} \)
indicates that there is a dependence from \( i \) to \( j \), then \( T_p(i) \), the execution of \( i \),
must be executed before \( T_p(j) \), the execution of \( j \):

\[
\forall i, j, p, q, \quad i \rightarrow j \in d_{pq} \Rightarrow T_p(i) \prec T_q(j)
\]  

(3.3) where \( \prec \) is the lexicographic ordering operator. This is to be confirmed by the
following computation (3.4), with (3.5) to ensure the mapping is 1-1 so that
the new program performs exactly the same set of computation as the original
program.

\[
\exists i, j, p, q, \quad s.t. \quad i \rightarrow j \in d_{pq} \land T_p(i) \succeq T_q(j)
\]  

(3.4)

\[
\forall i, j, p, q, \quad (p = q \land i = j) \iff T_p(i) = T_q(j)
\]  

(3.5)

### 3.3.4 Supporting facilities

Although UTF provides a uniform way to represent and reason about iteration
reordering transformations, it is not designed to decide which transformations
should be applied. This task is left to the surrounding systems. In order to
assist them in building and using mappings, UTF provides the following facilities
besides the legality test algorithm mentioned above.

Mapping component legality test: This is a supplement to the legality test de-
scribed above. A variable part \( V_p^k \) of the loop components in a mapping
is legal for a statement \( s_p \) if using it as a mapping component would not
violate any self-dependences on $s_p$. More formally, $V^k_p$ for $s_p$ is legal at level $k$ if and only if:

$$\forall i, j, \quad i \to j \in d^k_p \Rightarrow v^k_p(i) \leq v^k_p(j)$$

(3.6)

where $v^k_p : [i^1_p, \ldots, i^n_p] \to [V^k_p]$.

Mapping alignment algorithm: When a mapping is being constructed in a level by level manner, it is the variable parts of its loop components that are decided first, then this alignment algorithm supplements the constant part to ensure they align with each other in order to preserve the semantics of the code.

Code generation algorithm: The algorithm[65][66] takes a mapping and produces corresponding code in a level by level, recursive manner. It is capable of producing code that avoids and/or eliminates many of the guards that may occur around the statements when performing reordering transformations. When a schedule is provided, it can generate the code in one step, rather than in the traditional fixed phase order manner which applies one transformation at a time.

### 3.4 The Search Space

#### 3.4.1 Properties of UTF

It is worth noting that in a mapping, its loop components are always interleaved from each other by its syntactic components which correspond to the syntax tree structure. Therefore, for a statement in an $n$-nested loop, its mapping has $2n+1$ levels in total, in which there are $n$ loop components in odd-numbered levels, and $n+1$ syntactic components in even-numbered levels. For example, for the two statements in the double nested loops given in Figure 3.3, their identity mappings have two loop and three syntactic components each, as shown in $T_0$ and $T_1$.

Notice that none of the transformations, except loop tiling, changes a mapping’s depth if applied, regardless of the number of times it is applied. As for loop tiling, if applied to a single loop once, it increases the mapping depth by two (one for the loop component and one for the syntactic component), by four if to a double loop (two for loop and two for syntactic), etc. For instance, the schedule of tiling the double loop in Figure 3.3 with tile size 32 and 16 is demonstrated in Figure 3.6, together with the resulting code.
for (ti=0; ti<1024; ti+=32)
for (tj=0; tj<2048; tj+=16)
for (i = ti; i<ti+32; i++)
    for (j = tj; j<tj+16; j++) {
        0: b[i][j] = c[i] + d[j];
        1: a[i][j] = c[j] + d[j];
    }

$T_0: [i,j] \rightarrow [0, 32^k(i \text{ div } 32), 0, 16^k(j \text{ div } 16), 0, i, 0, j, 0]$
$T_1: [i,j] \rightarrow [0, 32^k(i \text{ div } 32), 0, 16^k(j \text{ div } 16), 0, i, 0, j, 1]$

Figure 3.6: Example of loop tiling

Because a schedule is considered equivalent to a sequence of transformations, a compiler can implement it by successively picking the transformation from the sequence in order and applying it. The mappings are updated accordingly at each step, their depths increase if loop tiling is picked, or remain unchanged otherwise. If loop tiling is applied repetitively, each time it is applied, the mapping’s depth increases by the same number. Therefore, the depth of the final mappings depends on the number of times loop tiling is applied.

The maximum number of times loop tiling can be applied depends on both the chosen tile size and the original loop size. Obviously the maximum loop depth is obtained if the compiler repeatedly tiles the loop with the smallest tile size 2. It is $\log_2 s$ where s is the size of the loop. For nested loop cases, the maximum loop depth is $\sum_{i=0}^{n-1} \log_2(s_i)$ where n is the number of loops nested and $s_i$ the size of the $i$-th loop.

For example, the maximum loop depth for the mapping of the loop given in Figure 3.3 is $\log_2 1024 + \log_2 2048 = 21$, the corresponding syntactic and mapping depth are $21+1 = 22$ and $21+22 = 43$ respectively.

By using the operators $\text{div}$ and $\text{mod}$ for loop tiling, UTF makes the affine functions in the mapping’s loop component more complex and difficult to manipulate, as shown in [65]. This problem is compounded by repetitive tiling. For simplicity, new temporary loop variables are introduced to modify the UTF notation. This is better explained by considering the new variable(s) introduced in the code generated. These new temporary variables are all attached with their corresponding tile size(s). For example, mapping $T_1$: $[i,j] \rightarrow [0, t_i^{32}, 0, t_j^{16}, 0, i, 0, j, 0]$ represents the transformation of tiling the nested loop in Figure 3.3, where $t_i^{32}$ is the new variable (tempi in Figure 3.6) introduced by tiling the $i$-loop with tile size 32, and $t_j^{16}$ is the new variable (tempj in Figure 3.6) introduced by tiling the $j$-loop with tile size 16.
This modification removes the operators \( \text{div} \) and \( \text{mod} \) from the affine functions and makes them tidier and easier to manipulate. However, it introduces additional tests in condition \( C \), for example those used to ensure the iteration space sizes remain intact after tiling. They are added as legality constraints to the system. This way the burden of testing is migrated to the supporting system which allows the mappings to appear in a concise manner.

### 3.4.2 A closer look

With the introduction of temporal variables, the loop components of a mapping can be represented simply as a linear function of the loop variables (both originals and derived temporaries). If an original loop variable is renamed for convenience as \( i_{k,0} \) and its derivation \( i_{k,1}^{t_1}, i_{k,2}^{t_2}, \ldots, i_{k,n}^{t_n} \), another general form of mapping can be obtained as below:

\[
T : [i_0, \ldots, i_m] \rightarrow [c_0, l_1, c_2, l_3, \ldots, l_{d-1}, c_d] : C \tag{3.7}
\]

where

\[
\forall x \in \{1, 3, \ldots, d-1\}, l_x = c_x + \sum_{m=0}^{s} \sum_{n=0}^{t_m} (c_{x,m} * l_{m,n}^n), \tag{3.8}
\]

in which all \( c_x \) and \( c_{x,m} \) are integer constants.

Recall that the loop components, \( l_x \)s (\( \forall x \in \{1, 3, \ldots, d-1\} \)), represent the syntactic part of the mapping and are potentially linear combination of surrounding iteration variables. The variables \( r_x \)s are loop variables such as the \( t_i^{32}, t_j^{16}, i \) and \( j \) as shown before, \( c_x \)s (\( \forall x \in [0, d] \)) and \( c_{x,m} \)s (\( \forall x \in \{1, 3, \ldots, d-1\}, m \in [0, s] \)) are their coefficients in the linear function, and \( t_m \)s (\( m \in [0, s] \)) are their tile size coefficients. All the coefficients are integers which vary in a relatively small range. \( C \) includes some extra constraints on them as mentioned above.

The combination of all these coefficients forms a huge space to explore. Notice that this mapping is just for one single statement in the original loop, such as statement 1 in Figure 3.3. If the loop contains \( s \) statements, all possible mapping combinations must be considered. This introduces \( s-1 \) times more coefficients into consideration. In addition, each of the legality constraints (as given in Equation (3.3), (3.4) and (3.5)) must be satisfied at a certain level to ensure the schedule is legal. The variety of such constraints (in both the number of them and the levels at which they are satisfied) adds extra dimensions to the optimisation space.

Every single point in such a space represents a potentially legal schedule for the loop. Many of them obviously appear illegal or unlikely to be beneficial, but without prior knowledge of the loop and other information, it is impossible to
know in advance where these bad points are and to prune them before the search starts.

### 3.4.3 A naive exhaustive scan algorithm

This thesis is concerned with effectively searching this UTF-based optimisation space. There are a large number of uncertainties in this space, for example, the mapping depth, the tile sizes, the coefficients in the linear functions of the loop components, and the syntactic components. If each of these uncertainties is considered as a separate dimension of the search space, which has its own range of valid values, the search space can be considered as a polyhedron.

As explained before, there exist various legality constraints imposed on these uncertainties. In addition, some of the uncertainties may affect the range of some others, for instance, the range of tile sizes may affect the mapping depth, and vice versa. All these constraints make the search space polyhedron highly irregular and difficult to specify in a descriptive manner.

We wish to develop a naive exhaustive scan algorithm. This algorithm is capable of scanning, in an exhaustive manner, a polyhedral space which is a superset of the target search space, but more regular and easier to manipulate. In this space, there exist points outside the target space, which will be eliminated via the legality constraints, there may also exist different points referring to the same transformation. For example, schedules \((T_0, T_i)\) and \((T_0', T_i')\) in Figure 3.7 are different points in the space, but they both represent applying loop interchange to the code segment in Figure 3.3. However, neither of these two cases affects the correctness of the exhaustive scan algorithm, as the legality test and code generation provided by UTF can eliminate them. Practically considering the number of points in the space and the cost to carry out all the legality tests involved, the cost of such a scan is very high, but it does provide a way to specify the target search space and estimate its size.

The naive exhaustive scan algorithm divides the scanning of the polyhedron into three phases, each dealing with a sub-polyhedron of a lower dimension. Each single point in the sub-polyhedron of the previous phase represents a polyhedron which the following phase has to explore.

Phase 1: This phase scans the space of schedule loop depths. In a nested loop containing \(n\) statements, the loop depth of each statement \(s_i\) may vary within \([l, u]\) where constant \(l\) corresponds to the original loop depth and \(u\) the loop depth in the most extreme case as discussed above. This phase
for (j = 0; j < 2048; j++)
    for (i = 0; i < 1024; i++) {
        0: b[i][j] = c[i] + d[j];
        1: a[i][j] = c[j] + d[j];
    }

$T_0: [i, j] \rightarrow [0, j, 0, i, 0]$
$T_1: [i, j] \rightarrow [0, j, 0, i, 1]$

$T'_0: [i, j] \rightarrow [0, j, 0, i, 0]$
$T'_1: [i, j] \rightarrow [0, j, 0, i, 2]$

Figure 3.7: Different schedules representing the same transformation

of scanning tries every possible combination of the statements’ mapping depth. The size of this polyhedron is therefore

$$size_1 = \prod_{i=1}^{n} (u - l + 1) = (u - l + 1)^n$$  \hspace{1cm} (3.9)

For example, for the nested loop in Figure 3.3, the number of points to scan is $(21 - 2 + 1)^2 = 400$.

Phase 2: For each point obtained from the first phase scan, the second phase deals with the space of level(s) where the constraints (in the form of Equation (3.3) and (3.4)) are to be satisfied, either at syntactic or at loop components of the corresponding mappings. Supposing there are $c$ constraints in total, each dependency $D_k : s_i - s_j$ is satisfied at mapping level $d_k$ of the corresponding mappings of statement $s_i$ and $s_j$, where $d_k$ may vary from 0 to $v_k = \min(m_i, m_j)$ ($m_i$ and $m_j$ are mapping depths of $s_i$ and $s_j$ respectively). The size of this polyhedron at this scan phase is

$$size_2 = \prod_{k=1}^{c} v_k$$  \hspace{1cm} (3.10)

Phase 3: Each point obtained from the second phase scan stands for a complete schedule whose coefficients (as shown in Equation (3.7) and (3.8)) are still to be decided. The number of these coefficients (for convenience, $d$) is known, although large, so there is a $d$-dimension sub-polyhedron to explore, with each coefficient varying within a certain range. Scanning this polyhedron is trivial.

Each point in this sub-polyhedron represents a potential schedule which, if passing all the legality tests, is a valid schedule a compiler should consider.
If \([l_c, u_c]\) is the common range of all \(s_c\) coefficients \(c_r s\) and \(c_s s\) in Equation (3.8), and \([l_t, u_t]\) is the common range of all \(s_t\) coefficients \(t_s s\) in Equation (3.8), the dimension of this sub-polyhedron is \(s_c + s_t\), and the size is:

\[
\text{size3} = (u_c - l_c + 1)^s_c \times (u_t - l_t + 1)^s_t
\]

(3.11)

Given enough time, the above three-phased scan algorithm can exhaustively visit all the points in the search space this thesis tends to explore. It provides an estimation of the size of the space as

\[
\text{size} = \text{size1} \times \text{size2} \times \text{size3}.
\]

(3.12)

Consider the code segment given in Figure 3.3, if tile sizes are allowed to vary from 1 to 10, unrolling factors from 1 to 20, the coefficient \(c_r\) from -5 to 5, \(c_{x,m}\) from -3 to 3, there are roughly \(1.515\text{E}29\) cases to be considered in total. Clearly, any realistic search algorithm will have to drastically reduce the number of points considered.

### 3.5 Features and Extension

In order to describe both the program and the optimisation space to search, a compiler usually concentrates on a set of features from the target code and the runtime environment, for example loop nest depth and array dimension. These features abstract the important details of the code. They enable a compiler to classify the codes into various categories and store the corresponding optimisations accordingly for retrieval, as shown in [86].

In order to classify the programs encountered and retrieve the results of previous efforts, this thesis uses the following set of features. They are grouped into three types: memory access, loop structure and code respectively. The features marked with asterisks can be obtained from UTF, whilst the others are used as its extensions. It is noticed that no feature about the runtime environment is included, as this is beyond the scope of the thesis.

1. memory access
   
   (a) number of array references
   (b) number of arrays used
   (c) linear array access (True/False)
   (d) number of array elements reuses across iteration *
(e) uniform data dependency (True/False) *

2. loop structure

(a) loop nest depth *
(b) loop sizes *
(c) perfectly nested loop(s) (True/False) *
(d) loop step(s) *
(e) abnormal exit(s) (True/False)

3. code

(a) number of arithmetic operations
(b) number of method calls
(c) conditional control structure within the loop (True/False)
(d) number of statements within the loop nest *

The above set of features is smaller than the set of performance-related indices used in [86]. It is worth noting that although the classification could be more precise if more features/indices are used, it comes with a higher cost for the classification and retrieval efforts. There is a tradeoff to be made between these two key factors, which deserves further exploration.

3.6 Summary

This chapter has briefly discussed the main technical challenges to be faced. A short review of UTF has been provided, before its mapping notation was used to explicitly specify an optimisation space via a naive exhaustive scan algorithm. An estimation of the optimisation space size has been given by way of an example. An introduction was then given to the extension made to UTF for the compiler to classify program features that may later be correlated against program optimisations.
Chapter 4
Heuristic Search

4.1 Introduction

Although the exhaustive scan algorithm described in the previous chapter can cover all the potential points in the target search space, it is infeasible for an optimising compiler to use such an exhaustive search, due to the size of the space and the associated cost of legality test and code generation.

It may be possible to reduce the optimisation space by exploring the constraints among the parameters to a greater depth. However, in order to eliminate the points outside the target search space and the duplicate points representing the same schedule, many more constraints would be introduced. This would make the space less regular and the cost of legality testing in UTF even higher. A reasonable alternative is a more efficient search algorithm, which uses prior results and heuristics to direct its search, in order to locate good points quickly.

This chapter presents a new heuristic search approach which combines random sampling and heuristics to explore the optimisation space. It is able to locate good points quickly, as demonstrated later in chapter 6. The strategy of this search algorithm is introduced in section 4.2, followed by a brief summary of the algorithm and its supporting facilities in section 4.3. Sections 4.4 and 4.5 discuss the algorithm in depth. A brief and informal discussion of the coverage of the algorithm is given in section 4.6, before section 4.7 makes a comparison between this search algorithm and other algorithms used by related projects introduced in chapter 2.

4.2 Search Strategy

The main targets of the heuristic search algorithm are efficiency, flexibility and coverage. By efficiency, the algorithm is expected to find good points quickly in
the huge search space. By flexibility, it is expected to demonstrate the ability to adjust its search according to the information it collects during the search process, especially the latest information. This flexibility will ultimately improve efficiency. By coverage, the algorithm is expected not to neglect a point which should be included in the space. The search algorithm must make a careful balance between efficiency and coverage.

4.2.1 Additional notations

The loop and syntactic components of a mapping are grouped into two vectors named loop vector and syntactic vector respectively. For example, given a mapping \((c_0, l_1, c_2, l_3, \ldots, l_{d-1}, c_d)\), its loop vector is \((l_1, l_3, \ldots, l_{d-1})\) and its syntactic vector is \((c_0, c_2, \ldots, c_d)\). A mapping can be constructed by interleaving the elements of these two vectors.

As explained before, the syntactic vector is a vector of integer constants, and the loop vector is a vector of linear functions of all the original or derived iteration variables, namely \(i_0, i_1, \ldots\) and \(i_x\). Loop vector \(LV\) can be presented as \(LV = I \times M\) where \(I\) is an \((x+2)\)-element vector of the iteration variables and \(M\) an \((x+2) \times (x+1)\) matrix of integer constants as shown below:

\[
I = (i_0, i_1, \ldots, i_x, 1)
\]

\[
M = 
\begin{pmatrix}
c_{0,0} & c_{1,0} & \cdots & c_{x,0} \\
c_{0,1} & c_{1,1} & \cdots & c_{x,1} \\
\vdots & \vdots & \ddots & \vdots \\
c_{0,x} & c_{1,x} & \cdots & c_{x,x} \\
c_0 & c_1 & \cdots & c_x
\end{pmatrix}
\]

Given a loop vector \(LV\), its default schedule transforms the code block where all the statements in the loop(s) remain in their original positions. Each statement is assigned a mapping whose loop vector is \(LV\) and syntactic vector represents its position in the original loop nest. Intuitively, the default schedule considers the loop body as containing only one compound statement which consists of all the statements in the original loop body. Therefore, transformations such as loop distribution, fusion and statement reordering are not included in the default schedule.

For instance, if we have a loop vector \(LV = (l_0, l_1, l_2)\) for the example in Figure 3.3, the mappings assigned to the two statements have syntactic vectors \((0, 0, 0, 0)\) and \((0, 0, 0, 1)\) respectively. This leads to the default schedule as shown in Figure 4.1.

58
\[ T_0: [i, j] \rightarrow [0, l_0, 0, l_1, 0, l_2, 0] \]
\[ T_1: [i, j] \rightarrow [0, l_0, 0, l_1, 0, l_2, 1] \]

Figure 4.1: An example of default schedule

4.2.2 Mapping construction as a two phase process

Using the above notations, mapping construction can be considered as a two phase process. In the first phase, the iteration variable vector \( I \) and the matrix \( M \) are decided and the loop vector \( LV \) is then obtained by \( LV = I \times M \). The second phase decides the syntactic vector \( SV \). The mapping is then constructed by interleaving the elements of \( SV \) and \( LV \).

UTF requires that each statement in the loop be assigned a separate mapping. Theoretically, their mappings do not necessarily share a common loop vector as in the default schedule case discussed above. In such cases, for the mapping of each statement, its loop vector must be decided separately, before its syntactic vector.

4.2.3 Random

With the help of the above notations, the task of optimisation space exploration is transformed into searching for good points in a large parameter space where parameters are from both the syntactic and loop vector(s).

Many approaches have been developed for searching a parameterised space, as discussed in chapter 2. Using randomisation has proved useful in [50] and other projects. Considering there is no prior knowledge of the search space, randomisation is considered a realistic starting point for the heuristic search algorithm.

4.2.4 Rotation policy

The separation of loop and syntactic components divides the target search space into two subspaces associated with the loop and syntactic vectors, which are explored by \( L\text{-Search} \) and \( S\text{-Search} \) respectively.

The L-Search focuses on loop vector subspace exploration. By default, it generates various loop vectors, and tests their default schedules. The S-Search explores the syntactic vector subspace. By default, it picks out a good loop vector and tests various schedules constructed by combining the loop vector with various syntactic vectors.

Since the L-Search and S-Search focus on different aspects of mapping construction, neither of them shall take sole control of the search process permanently.
A rotation policy is adopted in which they are given and relinquish control in turn. In each round, L-Search or S-Search checks a number of points in the space and evaluate those legal points, before relinquishing the control.

A variant of this policy could also be used, such as deciding, after a round of either L-Search or S-Search, which of them has the greater potential to find good points more quickly. The result of this evaluation will be used to bias future searches. In this case, there is a trade-off to be made between efficiency and coverage.

4.2.5 Loop first, then syntactic

During the search process, the heuristic search algorithm should aim to determine where the search should focus on, based on the collected profile information. Such results are used as a heuristic to increase the profitability. This is one aspect of the flexibility that the heuristic search algorithm aims to achieve. The variant of the rotation policy discussed above is an example of such flexibility.

Because the sizes of the loop and syntactic vector subspaces depend on both the code and the potential parameter ranges, it is difficult, if not impossible, to judge in advance which of them is larger. In general, there are more varieties in the loop vector subspace than in the syntactic vector subspace. This is because it contains a larger number of parameters. As explained later in section 4.4 and 4.5, these parameters and linear functions are linked with the transformations such as loop tiling, loop unrolling, loop skewing, loop reversal, loop alignment, etc. There are more varieties in the combinations of these transformations than those of the transformations linked with the syntactic vector subspace, such as loop fusion, loop distribution, statement reordering, etc.

Based on this observation and previous experience[80][128], it is believed that there are likely to be more opportunities for performance improvement within the loop vector subspace than in the syntactic vector subspace. Therefore, the heuristic search algorithm should try to decide the loop vector first before considering the variety of syntactic vectors. This overall search process starts with an L-Search attempt.

4.2.6 Simple first

Although the effectiveness of random searching has been demonstrated in small regular search spaces[50], its behaviour remains unknown in the much larger space considered in this thesis. As the interference among transformations is complex, the density of good points within this space is hard to predict. Considering the
size of the space, it is reasonable to make a sceptical estimation that the density is very low. Therefore, it is less likely that a plain random search can find good points within a small number of attempts. In order to succeed, the random search must adopt some heuristics to achieve acceptable efficiency.

Previous experience\cite{38,118,72,80,120} shows that an optimising compiler seldom needs to apply many available transformations to the code in order to achieve significant performance improvement. In most cases, the majority of improvement comes from either one or a small number of transformations. As a mapping can be understood as an abstraction of a traditional transformation sequence, a "simple mapping" intuitively stands for either a single transformation that is more likely to bring performance improvement in many cases, according to previous experience; or a short transformation sequence, if more than one transformation is included. Based on these observations, the heuristic search algorithm should try simple mappings before considering more complex ones.

This "simple first" philosophy is a desirable complement to the random strategy discussed before.

4.2.7 Explore to depth if a good hint is found

Inspired by the experimental results of \cite{70,72}, the proposed heuristic search algorithm is expected to demonstrate similar flexibility. If a good point is found, it should be able to explore the surrounding subspace where even better points may reside. This flexibility is a desirable addition to the heuristic search algorithm.

There is a balance to be maintained between flexibility, efficiency and coverage. For instance, it is desirable that when the search algorithm finds a good schedule, it can decide by itself whether or not it would be more effective to derive other schedules from it. But it is equally important that, if the search algorithm finds that this in-depth search does not bring encouraging results, it can decide by itself to quit and launch a new search elsewhere. This idea is similar to the "hill climbing" technique used in \cite{38}.

4.3 Heuristic Search Algorithm

4.3.1 Bias mechanism

A series of decisions have to be made at each iteration of the optimisation space search. For each of them, the available options are either statically decided by the search algorithm itself, or obtained from run-time information. The decision is then made randomly from these options, as explained above.
A bias mechanism must be introduced because the heuristic search expects good options to have a greater chance of being chosen. The available options are assigned different weights by the decision makers according to the biases decided by the search algorithm. The Roulette Wheel Parent Selection[40] is a biased random selection algorithm widely used in genetic algorithms, which, on the whole, gives more reproduction chances to those organisms that are the most fit. It is used to implement the bias mechanism in all the decision makers considered in this thesis. In addition, all these biased decision makers should periodically review their biases (the weights of their options) in order to adjust themselves to the latest progress and to avoid exploring too deep in one small subspace whilst leaving the rest unexamined. Furthermore, together with a redundancy check, this periodically-reviewed bias mechanism helps to ensure the coverage of the heuristic search algorithm, as explained later in section 4.6.

4.3.2 Supporting facilities

A successful search algorithm needs some supporting facilities, which are summarised in this subsection.

A profile database is needed to store, retrieve and analyse profile information collected during the search process. For the search algorithm, there are two major types of profile information to be retained and used in the profile database.

The first type is called schedule profile. Each database entry contains a unique schedule, along with its performance and other relevant information. Schedule profiles are stored in the S-pool of the profile database. As there may exist different schedules leading to identical code, the S-pool is expected to provide a redundancy check for each entry it receives, in order to ensure that each entry is unique.

The other type of profile is called loop profile. Each entry contains a loop vector and its contributions (i.e. how well it improves the performance) in different schedules. Loop profiles are stored in the L-pool of the profile database, which also provides facilities to pick good candidate loop vectors for the S-Search to try.

Other supporting facilities include a random number generator, random vector generator and random matrix generator. They are expected to generate random outputs which are consistent with the bias given to them as input. This enables the search algorithm to explore the optimisation space in a random but biased manner.
4.3.3 The algorithm

The heuristic search algorithm contains three major parts, the L-Search, S-Search and steering module. They are briefly introduced in this subsection.

4.3.3.1 L-Search

The L-Search aims to explore the subspace corresponding to the mapping’s loop vector variety. It generates a certain number of loop vectors and evaluates them using their defaults schedules. The evaluation results are stored in the profile database. A detailed description of the L-Search is given in section 4.4, which explains each step in depth along with its variants.

4.3.3.2 S-Search

Whilst the L-Search focuses on the loop component subspace, the S-Search aims to explore the subspace concerned with the syntactic vector varieties. It chooses a good loop vector from the L-pool and combines it with various non-default syntactic matrices generated in order to construct new schedules. These schedules are evaluated and the results are saved in the profile database. A detailed description of the S-Search is given in section 4.5, which explains each step in depth and discusses its variants.

4.3.3.3 Steering module

The L-Search and S-Search focus on different aspects of the optimisation space. They are coordinated by a steering module during the search process. The steering module is summarised using the pseudo code shown in Figure 4.2, which demonstrates the control flow of the heuristic search algorithm.

By default, the steering module tries these two searches in a rotation manner. However, it is also expected to demonstrate flexibility when necessary. For instance, if the L-Search or S-Search find very encouraging results, it is preferable to keep on exploring the corresponding subspace. In order to achieve this "explore to depth if a good hint is found" flexibility, both the L-Search and S-Search should inform the steering module of their results during the search process.

It is worth noting that this flexibility may make the search inefficient in certain cases. For instance, some particularly good points found in one L-Search round could make the decision maker select the L-Search for many more rounds. However, if no more good points can be found, it is not worth exploring this subspace further. In such cases, a hill-climbing-like technique is needed for the
initialisation;

try 1st round of L-Search;
evaluate the achievement of L-Search;

try 1st round of S-Search;
evaluate the achievements of S-Search;

repeat   // when optimisation budget allows
{
   make a decision for the next round of the heuristic search;
   //based on the results of achievement evaluations

   if (decision is "try L-Search") {
      try L-Search;
      evaluate the achievement of L-Search;
   }
   else {   // the decision is "try S-Search"
      try S-Search;
      evaluate the achievement of S-Search;
   }
}
}

Figure 4.2: Pseudo code of the steering module

The steering module to revert to the S-Search, despite the particularly good points
found before.

In order to achieve this desirable flexibility but avoid its negative impact, the
decision maker should consider both the latest and the overall achievements of
both searches and make a choice by carefully balancing these factors, favouring
the latest information collected during the search process.

4.4  L-Search

4.4.1  Introduction

The L-Search aims to explore the subspace corresponding to the variety in loop
vectors. By default, each round of the L-Search repeats for a certain number of
times the following steps, each of which will be explained in depth in the following
subsections.

1. construct a loop vector $LV$;

2. construct the default syntactic matrix $SV$;

64
3. construct the default schedule for $LV$
4. test the schedule;
5. store the profile into the profile database.

4.4.2 Construct a loop vector

Both the iteration variable vector $I$ and transformation matrix $M$ must be decided before the loop vector $LV$ is obtained via $LV = I \times M$.

The iteration variable vector $I = (i_0, i_1, ... i_x, 1)$ consists of both the original and derived iteration variables. The dimensions of its sub-vector $(i_0, i_1, ... i_x)$ are called loop depth in the heuristic search algorithm. Intuitively, it stands for the depth of the nested loop generated. This must be decided first, in order to construct an $I$ and $M$ of the correct dimension(s).

4.4.2.1 decide tile size(s)

Loop tiling is the only transformation that introduces derived loop variables. One new derived loop variable is introduced by applying tiling to one loop once, regardless of the tile size. Most optimising compilers apply tiling just once. However, with the presence of a multi-level cache hierarchy, performance could be further improved if loop tiling is applied repeatedly. The number of times tiling can be applied depends on the tile sizes chosen and the original loop size(s), as demonstrated in section 3.4. Therefore, tile depth, the number of times a given loop is tiled, may be an arbitrary integer within a certain range.

Because each loop in the original loop nest could be tiled a different number of times, they must be considered independently. For each of them, there are three decisions to be made: whether to tile it; how many times it should be tiled, i.e. tile depth $td$; and what the tile size should be. The first two decisions are made by a LoopDepthDecisionMaker and the third one is made by a TileSizeSelector.

For each loop in the loop nest, before the LoopDepthDecisionMaker chooses a loop depth randomly, it must calculate the whole range of possible loop depths, from 1 (no tiling is applied) to $\log n$ ($n$ is the original loop size), and assign a different weight to each value. Following the "simple first" strategy, it biases to 1 and 2, which intuitively mean no loop tiling is applied and it is applied once respectively. The overall loop depth for the whole loop nest is the sum of the loop depths chosen for each loop in it.

Once the loop depth is decided, the TileSizeSelector will then produce a tile size array for the loop by randomly picking a number from a range of integer
// input: 1) n: original loop depth;
// 2) s[n]: original loop size array;
// output: ts[n][]: tile size array;

repeat // for each original loop depth from 0 to n-1
{
    select a loop depth ld;
    repeat // for each loop depth 1 to ld
    {
        select ts; // the tile size, potential range [1, s[i]]
        assign ts to ts[i][j];
        set s[i] to ts;
    }
}

Figure 4.3: Pseudo code for deciding tile sizes

candidates. Notice that after each round of loop tiling, the size of the newly
generated loop is the tile size chosen in the previous round, therefore the range
of available tile sizes changes accordingly. This procedure can be summarised as
pseudo code given in Figure 4.3.

After loop tiling, there are \( m=\sum_{i=0}^{n-1}(1 + ts[i].length) \) loops in the resulting
nested loop. The iteration variable vector \( I \) is \( (i_0, i_1, \ldots, i_{m-1}, 1) \). The problem that
now arises is how to distinguish the derived iteration variables from the original
ones. This is solved by the following rules:

1. Starting from the left, the \((ts[i].length)\) variables are iteration variables de-
   rived from tiling the original \( i \)-th loop, where \( i \) is in the range 0 to \( n-1 \).

2. The \( n \) rightmost variables are new names given to the original iteration
   variables in the same order as they appear in the code.

For example, given the loop nest in Figure 3.3, the tile size array \( ts = \{\{8\},
\{16, 4\}\} \) indicates that the outer loop \( i \) is to be tiled once with tile size 8, and
loop \( j \) be tiled twice, first with tile size 16 and the second time with tile size 4.
This results in the new loop nest shown in Figure 4.4. The overall loop depth is
\( m=\sum_{i=0}^{n-1}(1 + ts[i].length) \) = 5. The iteration variable vector \( I \) is \( (i_0, i_1, i_2, i_3, i_4, 1) \). Using
the above approach, we can see that \( i_0 \) is the derived variable introduced by tiling
the original \( i \)-loop; \( i_1 \) and \( i_2 \) are derived from tiling the \( j \)-loop. For convenience,
they are given new names \( j_0 \) and \( j_1 \) in the Figure 4.4; and \( i_3 \) and \( i_4 \) are the new
names given to the original loop variables \( i \) and \( j \) respectively.
// original code segment (as in Figure 3.3)

//
// for (int i = 0; i<1024; i++)
//   for (int j = 0; j<2048; j++) {
//       0: b[i][j] = c[i] + d[j];
//       1: a[i][j] = c[j] + d[j];
//   }
//
// after loop tiling with tile size array {{8}, {16, 4}}

for (int i0 = 0; i0<1024; i0+=8)
  for (int j0 = 0; j0<2048; j0+=16)
    for (int j1 = j0; j1<j0+16; j1+=4)
      for (int i = i0; i<i0+8; i++)
        for (int j = j1; j<j1+4; j++) {
          0: b[i][j] = c[i] + d[j];
          1: a[i][j] = c[j] + d[j];
        }

Figure 4.4: Code after loop tiling

4.4.2.2 decide unrolling factor(s)

Loop unrolling is a transformation that UTF can represent via index set splitting. It replicates the body of the loop for a certain number (unrolling factor) of times in order to reduce the loop overhead and increase the instruction level parallelism by increasing the basic block size. Because it increases the number of statements in the resulting code, it is treated in a special way by the heuristic search algorithm.

Theoretically, all of the loops in the loop nest are potential candidates for loop unrolling. Similar to the loop tiling case, there are three decisions to be made in order: how many of them will be unrolled; which of them will be unrolled; and the unrolling factor(s). The first two decisions are made by an UnrollingDepthDecisionMaker and the last one by an UnrollingFactorSelector.

Following the "simple first" strategy, the UnrollingDepthDecisionMaker randomly selects from 1 to m (loop depth obtained before) the number of loops to be unrolled, with a heavy bias to 1. Intuitively, this means only one loop in the loop nest will be unrolled. It then randomly selects loop(s) to be unrolled, with a heavy bias to the innermost loop, for example, the j-loop in Figure 4.4. These two biases are consistent with the normal manner in which loop unrolling is applied.

Once the first two decisions are made, the UnrollingFactorSelector will randomly choose a valid unrolling factor for each loop to be unrolled. For the j-loop
/ input: 1) x: loop depth;
// 2) s[x]: loop size array;
// output: uf[]: unrolling factor array;
int index; // the index of the loop to be unrolled, range [0, x-1];
int factor; // the unrolling factor;

initialise uf[]; // no loop unrolling by default
decide num, the number of loop(s) to be unrolled;

repeat // for each loop 1 from 0 to num-1
{
    randomly select index;
    randomly select factor; // ranging from [1, s[1]]
    assign factor to uf[index];
}

Figure 4.5: Pseudo code for deciding unrolling factor(s)

for (int i0 = 0; i0<1024; i0+=8)
    for (int j0 = 0; j0<2048; j0+=16)
        for (int j1 = j0; j1<j0+16; j1+=4)
            for (int i = i0; i<i0+8; i++)
                for (int j = j1; j<j1+4; j+=2) {
                    0: b[i][j] = c[i] + d[j];
                    1: a[i][j] = c[j] + d[j];
                    2: b[i][j+1] = c[i] + d[j+1];
                    3: a[i][j+1] = c[j+1] + d[j+1];
                }

Figure 4.6: Code after loop unrolling

in Figure 4.4, the possible unrolling factors are 4, 3, 2 and 1 (1 means no unrolling is actually applied). This procedure is applied to all the loops picked out to be unrolled by the UnrollingDepthDecisionMaker. This procedure can be summarised as the pseudo code in Figure 4.5. For example, given the loop nest in Figure 4.4, the unrolling factor array uf[] = {1, 1, 1, 1, 2} means the innermost loop (j-loop) is to be unrolled with unrolling factor 2, which results in the new loop nest in Figure 4.6.

68
4.4.2.3 transformation matrix generation

With iteration variables \( i_0, i_1, \ldots \) and \( i_x \), the dimension of the iteration variable vector \( I = (i_0, i_1, \ldots, i_x, 1) \) is \( x+2 \). The transformation matrix \( M \) is therefore an integer matrix of size \((x+2) \times (x+1)\). Its transpose matrix \( M^T \) can be divided into two submatrices, i.e. \( M^T = (m \ a) \), where \( m \) is an \((x+1) \times (x+1)\) square matrix on the left and \( a \) is an \((x+1) \times 1\) matrix on the right. For example, for the loop nest in Figure 3.3, its iteration variable vector \( I \) and transformation matrix \( M \) are expressed as:

\[
I = (i, j, 1), \quad M^T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}
\]

(4.1)

where

\[
m = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad a = \begin{pmatrix} 0 \\ 0 \end{pmatrix}
\]

(4.2)

With these new notations, many of the transformations UTF considers can be represented by different \( M^T \)'s. For example, each matrix in the following Equations (4.3) to (4.6) represents a transformation to be applied to the sample loop nest in Figure 3.3.

**Loop Interchange** :
\[
M^T = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}
\]

(4.3)

**Loop Reversal** :
\[
M^T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}
\]

(4.4)

**Loop Skewing** :
\[
M^T = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}
\]

(4.5)

**Loop Scaling** :
\[
M^T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix}
\]

(4.6)

These transformations (loop interchange, loop reversal, loop skewing, loop scaling, etc.) and any arbitrary combination of them can be represented by a different matrix \( m \), because none of them changes the submatrix \( a \) of matrix \( M^T \), as demonstrated in the previous examples. Matrix \( a \) is used to represent loop alignment, as demonstrated in Equation (4.7).

**Loop Alignment** :
\[
M^T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}
\]

(4.7)

It is known that any arbitrary square matrix \( m \) can be obtained by, starting from an identity matrix, successively applying linear transformation to it via
matrix multiplication. Linear transformations can be represented by the submatrices $m$s such as those in Equations (4.3), (4.4), (4.5) and (4.6). In addition, any arbitrary $a$ can be obtained by starting from an all-0 one and arbitrarily assigning random values to its elements. Therefore, the matrix $M^T = (m, a)$ can present any arbitrary combination of the above transformations. This turns L-Search’s task of searching for a good mapping into searching for a good transformation matrix $M^T$. Considering the valid value range for each element (even within a small range) in $M^T$ and the number of elements in the matrix, the potential search space is huge.

In order to follow the "simple first" strategy, it is preferable that simple transformation matrices are generated before more complex ones. It is difficult to give a formal definition of a "simple matrix", and there is practically no need to draw a clear line between a simple matrix and a non-simple one. Intuitively, for the $(x+2) \times (x+1)$ matrix $M^T = (m \ a)$ considered here, it is considered simple if its submatrix $m$ is an identity matrix, or a matrix that can be obtained by applying one or a few steps of linear transformations on an identity matrix. Obviously, the above $M^T$ given in Equations (4.1), (4.3), (4.4), (4.5), (4.6) and (4.7) are all considered simple. Intuitively, the more $0$ elements a matrix has, the simpler it is. For example, the left matrix in Figure 4.7 is considered simpler than the right one.

In order to ensure that simple matrix is more likely to be created than more complex matrix, the heuristic search algorithm generates the transformation matrix $M^T = (m \ a)$ in the following way: starting from a default matrix (as shown in Figure 4.8) whose $m$ is an identity matrix and $a$ a zero matrix, it iteratively chooses one operation from all those available and applies it until a new matrix is produced. The available operations include basic linear transformations and assigning arbitrary values to $a$. They are randomly chosen in a biased manner in each step. In order to avoid generating identical matrices, every new matrix generated is stored in the profile database for a redundancy check. This procedure
\[
M^T = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]

Figure 4.8: An example of a default transformation matrix

// input: row and column of matrix to be generated;  
// output: the matrix M

create a default matrix M of size (row x column);  
repeat // when M is illegal
{
  repeat // if M has been generated before
  {
    // t: a matrix transformation
    randomly select t;
    randomly choose parameter(s) for t;
    apply t to M;
  }
}

Figure 4.9: Pseudo code for generating a random simple transformation matrix

can be briefly expressed as pseudo code in Figure 4.9. The decisions are made by biased random decision makers. This random matrix generation approach has been tested comprehensively. Even when asked to provide 1,000 different matrices, the matrices it generates are all considered simple enough, i.e. the transformation sequence it stands for can be recognised without much difficulty and obtained within a small number of steps. In addition, the matrices generated have a fairly even distribution, i.e. the chances are even for all elements in the matrix to obtain a new value. All these results show that this random matrix generation approach is fully capable of fulfilling the requirements.

4.4.2.4 bias in loop vector construction

It is worth noting that neither loop tiling nor loop unrolling is essential for code optimisation, so both deciding tile sizes and unrolling factors are optional steps during the loop vector construction. The whole procedure of loop vector construction is demonstrated by the pseudo code shown in Figure 4.10.
// input: 1) n: original loop depth;
//         2) ls[n]: loop size array;
// output: loop vector LV, which consists of
//         1) ts[[][]]: tile size array;
//         2) uf[]: unrolling factor array;
//         3) m: transformation matrix;

make a decision about loop vector construction;
if (decision includes tiling)
    decide ts;
else
    initialise ts to be an all-1 array;

if (decision includes unrolling)
    decide uf;
else
    initialise uf to be an all-1 array;

generate m;

construct loop vector LV with ts, uf and m;

Figure 4.10: Pseudo code of the loop vector construction
As previous work [80] has shown, in many cases tiling and unrolling are more likely to bring performance improvement than many other transformations. A *Loop Vector Construction Decision Maker* is used to make the decision, with bias to both of them.

### 4.4.2.5 Example

The loop vector $LV$ can be conceptually presented in a tuple form $(ts, uf, M)$, where $ts$ is the tile size array obtained from 4.4.2.1, $uf$ is the unrolling factor array obtained from 4.4.2.2, and $M$ is the transformation matrix obtained from 4.4.2.3. For example, with tile size array $ts = \{8\}$, \{16, 4\}, unrolling factor array $uf = \{1, 1, 1, 1, 2\}$, and the default transformation matrix $M$ as shown in Figure 4.8, the loop vector $LV$ is $(i_0, i_1, i_2, i_3, i_4, 1)$, which, when applied to the code in Figure 3.3, results in the code as shown in Figure 4.6.

### 4.4.3 Generate the default syntactic matrix

In order to construct the default schedule for loop vector $LV$, each statement in the loop nest must be assigned a separate syntactic vector, according to UTF. This is done by constructing a default syntactic matrix, each row of which stands for a syntactic vector for a statement.

The first question is how many statements are there in the resulting loop nest? This number is not necessarily the same as the number of statements, $s$, in the original loop nest, as loop unrolling replicates all statements in the loop body. The number of times the loop body is replicated is $u = \prod_{i=0}^{u^{length-1}} u f[i]$. Therefore the total number of statements in the resulting loop nest is $r = u \times s$, which is the row dimension of the syntactic matrix. The column dimension of the syntactic matrix is $x+1$, where $x$ is the number of original and derived iteration variables of the loop nest.

The procedure for the generation of the default syntactic matrix is illustrated by pseudo code as shown in Figure 4.11.

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 2 \\
0 & 0 & 0 & 0 & 3 \\
\end{pmatrix}
\]  

(4.8)

For example, Equation (4.8) shows the default syntactic matrix for the code segment in Figure 4.6. Another example of a default syntactic matrix is shown in Figure 4.12. It is generated for a loop vector composed of tile size array $ts = \{8\}$, \{16, 4\}, unrolling factor array $uf = \{1, 1, 1, 2, 2\}$ and a default
// input: 1) s: number of statements in the original loop nest;
//  2) uf: unrolling factor array;
// output: M: the default syntactic matrix;

// row: the number of statements in the resulting code
assign row the product of s and all the elements of uf;

// column: the number of iteration variables
assign column the size of uf;

initialise M to be an all-zero matrix of size (row x (column+1))

// if unrolling is applied to any but the innermost loop,
// then assign new values to these columns
repeat // for each column c, ranging from 1 to column-1
{
    assign temp the product of s and all elements uf[k];
    // where k ranges from (c-1) to (column-1)
    if (the (c-1)-th loop is to be unrolled)
    {
        assign (r / temp) to all M[r][c], where r ranges from 0 to row-1;
    }
}

// assign new values to elements in the rightmost column
assign (s * uf[column-1]) to temp;
assign (r % temp) to all M[r][column-1], where r ranges from 0 to row-1;

Figure 4.11: Pseudo code for generating the default syntactic matrix
transformation matrix, which intuitively means unrolling both the \(i\)- and \(j\)- loops in Figure 4.4 with unrolling factor 2. The corresponding default syntactic matrix and the resulting code segment are shown in Figure 4.12.

4.4.4 Construct the default schedule

With both loop and syntactic vectors decided, the last step is to allocate suitable syntactic vector(s) to statements in the original loop. Statement \(s_i\) (\(i = 0, 1, \ldots, u \times n - 1\)) is assigned the \(r\)-th row(s) of the syntactic matrix if \((r \% n = i)\), where \(n\) is the number of statements in the original loop. For instance, there are \(n = 2\) statements in the original loop nest in Figure 3.3, so rows 0 and 2 of the default syntactic matrix shown in Equation (4.8) are assigned to statement \(s_0\), and rows 1 and 3 are assigned to statement \(s_1\).

The default schedule is then obtained by combining these syntactic vectors with the loop vectors obtained in section 4.4.2. For example, the default syntactic matrix shown in Equation (4.8) results in the mapping for statement \(s_0\) as shown in Equation (4.9) and the mapping for statement \(s_1\) in Equation (4.10). The code generated is shown in Figure 4.6.

\[
T_0 : [i, j] \rightarrow \\
[0, i_0, 0, i_1, 0, i_2, 0, i_3, 0, i_4, 0] \cup \\
[0, i_0, 0, i_1, 0, i_2, 0, i_3, 0, i_4, 2] \tag{4.9}
\]

\[
T_1 : [i, j] \rightarrow \\
[0, i_0, 0, i_1, 0, i_2, 0, i_3, 0, i_4, 1] \cup \\
[0, i_0, 0, i_1, 0, i_2, 0, i_3, 0, i_4, 3] \tag{4.10}
\]

Similarly, for the syntactic matrix shown in Figure 4.12, rows 0, 2, 4 and 6 will be assigned to statement \(s_0\) of the original loop, and rows 1, 3, 5, 7 will be assigned to statement \(s_1\), which results in the code as shown in Figure 4.12.

4.4.5 Explore to further depth

The above steps discussed from 4.4.2 to 4.4.4 are integrated into the default approach of the L-Search, which is summarised in the algorithm in Figure 4.13.

In terms of flexibility and coverage, the heuristic search algorithm rotates control between the L-Search and S-Search. It is desirable that the L-Search can be equally flexible.
Syntactic Matrix

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 2 \\
0 & 0 & 0 & 0 & 0 & 3 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 2 \\
0 & 0 & 0 & 0 & 1 & 3 \\
\end{pmatrix}
\]

Resulting Code Segment

```java
for (int i0 = 0; i0<1024; i0+=8)
for (int j0 = 0; j0<2048; j0+=16)
for (int j1 = j0; j1<j0+16; j1+=4)
for (int i = i0; i<i0+8; i+=2) {
    for (int j = j1; j<j1+4; j+=2) {
        0: b[i][j] = c[i] + d[j];
        1: a[i][j] = c[j] + d[j];
        2: b[i][j+1] = c[i] + d[j+1];
        3: a[i][j+1] = c[j+1] + d[j+1];
    }
}
for (int j = j1; j<j1+4; j+=2) {
    0: b[i+1][j] = c[i+1] + d[j];
    1: a[i+1][j] = c[j] + d[j];
    2: b[i+1][j+1] = c[i+1] + d[j+1];
    3: a[i+1][j+1] = c[j+1] + d[j+1];
}
```

Figure 4.12: Example of a default syntactic matrix and the resulting code
// input: 1) n: original loop depth;
// 2) ls[n]: original loop size array;
// 3) s: number of statements in the original code;
int[][] ts; // tile size array
int[] uf; // unrolling factor array
Matrix dtm; // default transformation matrix
LoopVector lv; // loop vector
Matrix tm; // transformation matrix
Matrix dsm; // default syntactic matrix
Schedule schedule; // schedule

// try the 1st attempt
decide ts, uf and dtm;
construct lv from ts, uf and dtm;
construct dsm;
construct schedule with lv and m;
test schedule;
store the profile into the profile database;

// try more attempts
repeat // for a certain number of times
{
    derive new tm from dtm;
    construct lv from ts, uf and tm;
    construct schedule from lv and dsm;
test schedule;
store the profile into the profile database;
}

Figure 4.13: Pseudo code of the default L-Search
By default, each round of the L-Search tries a certain number of transformations. However, due to the random selection nature of the algorithm, the loop vectors tried in one round are irrelevant to those in other rounds. One desired flexibility is that if one loop vector is found to be significantly better than the others, or the majority of the others, the L-Search should bias its search to loop vectors similar to it. Two key issues must be addressed in order to provide this functionality. The first is how to evaluate this significant superiority in terms of performance. The other is how to obtain similar loop vectors. They are discussed below.

4.4.5.1 test the loop vector’s significant superiority

Previous experience[50][70][80] in iterative optimisation shows that it is rare to have merely one or a few loop vectors on top of the L-pool whose performances are well above the rest after a number of rounds of the L-Search. It is more likely that there are a number of loop vectors that could be similarly good and significantly better than the rest. If the average of all of them were used as the index of the good performance, it could be misleadingly raised by the performance of the significantly good ones, whose significance were therefore degraded. In order to avoid this, the loop vectors are kept in descending order according to the performance of their default schedules. The heuristic search algorithm chooses the average performance of only the lower section of them as the index instead. For instance, its current implementation picks the top twenty loop vectors from the L-pool, calculates the average performance of the lower half of them (eleventh to twentieth), and if the performance of the first loop vector is 20% better than this average, it is considered significantly good and worthy of further exploration.

The approach used by the heuristic search algorithm to identify significantly good loop vectors is summarised as pseudo code in Figure 4.14.

As the L-Search goes on, it is likely that there will be more loop vectors good enough to join the top list, and the average performance of the reference section will arise, which raises the significant good index. Only loop vectors even better than the best currently found will be considered significant good and worthy of further exploration. But if there is no such significantly good loop vector, the heuristic search algorithm will search elsewhere. Flexibility is therefore achieved.

4.4.5.2 construct similar loop vector(s)

Deriving similar loop vectors from an existing one is comparatively straightforward. The arrays of tile size(s) and unrolling factor(s) are kept untouched, and
// input: 1) L-pool;
// 2) s: size of the loop vectors with best performance;
// 3) t: threshold of significant superiority, e.g. 20%;
// 4) l: reference size of significant superiority, e.g. s/2;
LoopVector[s] tops; // array of top loop vectors

assign tops the loop vectors of top performance in L-pool;
  // in descending order of their performances

// avg: the reference performance of "normally good"
average the performance of loop vectors tops[s-1]..tops[s-1];
assign the result to avg;

// index: the reference performance of "significantly good"
assign (avg * (1+t)) to index;
if (the performance of tops[0] is greater than index)
  it is considered "significant good";
else
  it is considered "normally good";

Figure 4.14: Pseudo code for identifying "significantly good" loop vector

the new transformation matrices are derived using the same approach shown in
Figure 4.9, but starting from the transformation matrix of the chosen loop vector,
rather than from a default identity one. The new loop vectors can be constructed
in the same manner as specified before. The same default syntactic matrix is used
to generate default schedules for them. This to-depth variant of the L-Search is
summarised as pseudo code in Figure 4.15.

4.4.6 Bias in L-Search control

Due to the introduction of this to-depth variant, an LSearchDecisionMaker is used
to decide, at the start of every L-Search round, which of these two approaches
should be used, with bias to the default one. However, the first round should
always use the default approach, because it does not have any loop vector profile
in the profile database at the beginning of the heuristic search.

4.5 S-Search

The S-search aims to explore the syntactic vector subspace. Instead of starting
from scratch, it picks out loop vector(s) with good performance and combines
them with various syntactic matrices to construct new schedules. This process is
LoopVector lv; // the loop vector with best performance
Matrix m;    // the syntactic matrix
int[] ts;    // tile size array
int[] uf;    // unrolling factor array
Matrix t, tm; // the transformation matrix
Schedule schedule; // the new schedule

find lv, the loop vector of the best performance in L-pool;
pick out m, ts, uf and t from lv;

repeat // for a certain number of times
{
    generate tm from t;
    construct lv using ts, uf and tm;
    construct schedule from lv and m;
    test schedule;
    store the profile into the profile database;
}

Figure 4.15: Pseudo code of the to-depth L-Search

summarised below:

1. Choose \( n \) loop vectors from the L-pool, favouring those with good performance;

2. Generate \( n \) random syntactic vectors;

3. Construct all \( n \) mappings using the procedure given in the above subsection, so that a full schedule \( S \) is obtained;

4. Carry out a legality test on the schedule, if it fails, go back to step 1;

5. Check if \( S \) has been tested before, if yes, go back to step 1;

6. Otherwise, generate the code for \( S \);

7. Run the code and get the performance \( P \);

8. Store the profile entry \((S, P)\) in the S-pool, if it passes the redundancy test.

### 4.5.1 Choose loop vectors

Because of the "explore to depth if a good hint is found" strategy, the S-Search tends to choose the loop vectors that have been shown to be good before, in either...
Syntactic Matrix (default):

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]

Resulting Code

```java
for (int i0 = 0; i0<1024; i0+=8)
    for (int j0 = 0; j0<2048; j0+=16)
        for (int j1 = j0; j1<j0+16; j1+=4)
            for (int i = i0; i<i0+8; i++)
                for (int j = j1; j<j1+4; j++)
                    {  
                    0: b[i][j] = c[i] + d[j];  
                    1: a[i][j] = c[j] + d[j];  
                }
```

Figure 4.16: Syntactic matrix’s impact on resulting code (example.1)

the L-Search or S-Search. A simple approach is always to choose the best from all potential candidates. However, this approach may sometimes explore too deep in the wrong direction, i.e. if the combination of this best loop vector and different syntactic matrices does not bring further performance improvement, but it still remains the best of all the loop vectors, this leads to a staleness in the search. In order to avoid such a drawback, a variant of this approach is developed and used, which randomly selects one loop vector from a small number (for example, five or ten) of the top candidates, using a biased random selection algorithm.

### 4.5.2 Syntactic matrix generation

Intuitively, the syntactic vector is linked with transformations such as loop fusion, loop distribution and statement reordering, etc. If a loop vector is combined with different syntactic matrices to construct schedules, the resulting codes will be different in loop organisation but have all the statements undergoing the same transformation as the default schedule of the loop vector. For example, Figure 4.16, 4.17, 4.18 and 4.19 demonstrate how the original code segment shown in Figure 3.3 will be transformed. All these schedules share a common loop vector which is composed of a tile size array \( ts = \{8\}, \{16, 4\} \), an unrolling factor array \( uf = \{1, 1, 1, 1, 1\} \) and the \( 6 \times 5 \) default transformation matrix as shown in Figure 4.8, but have different syntactic matrices as shown respectively.

Theoretically, all the matrices of the right dimensions must be considered. For
Syntactic Matrix:

\[ \begin{pmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{pmatrix} \]

Resulting Code

```java
for (int i0 = 0; i0<1024; i0+=8)
  for (int j0 = 0; j0<2048; j0+=16) {
    for (int j1 = j0; j1<j0+16; j1+=4)
      for (int i = i0; i<i0+8; i++) {
        for (int j = j1; j<j1+4; j++)
          if (b[i][j] = c[i] + d[j];
          for (int j = j1; j<j1+4; j++)
            1: a[i][j] = c[j] + d[j];
    }
  }
```

Figure 4.17: Syntactic matrix’s impact on resulting code (example.2)

Syntactic Matrix:

\[ \begin{pmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0
\end{pmatrix} \]

Resulting Code

```java
for (int i0 = 0; i0<1024; i0+=8)
  for (int j0 = 0; j0<2048; j0+=16) {
    for (int j1 = j0; j1<j0+16; j1+=4)
      for (int i = i0; i<i0+8; i++) {
        for (int j = j1; j<j1+4; j++)
          0: b[i][j] = c[i] + d[j];
      for (int j = j1; j<j1+4; j++)
        1: a[i][j] = c[j] + d[j];
  }
```

Figure 4.18: Syntactic matrix’s impact on resulting code (example.3)
Syntactic Matrix:

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

Resulting Code

for (int i0 = 0; i0<1024; i0+=8) 
   for (int j0 = 0; j0<2048; j0+=16) 
      for (int i1 = j0; i1<i0+16; i1+=4) 
         for (int j1 = j0; j1<i1+16; j1+=4) 
            for (int j = j1; j<j1+4; j++) 
               1: a[i][j] = c[j] + d[j]; 
               0: b[i][j] = c[i] + d[j]; 
          }
   }

Figure 4.19: Syntactic matrix's impact on resulting code (example 4)

instance, all the \(2 \times 6\) matrices in the above examples. However, because statement
reordering may violate the dependency constraints imposed by the original code
and lead to semantically different codes, a legality test on the syntactic matrix
must be carried out to ensure that only legal schedules are generated.

Because the dependences are not known in advance for the heuristic search
algorithm, the chance of a randomly generated syntactic matrix being legal is
small. In order to generate a legal syntactic matrix, the S-Search uses a heuristic
random syntactic matrix generation algorithm, which is similar to the one used for
the random transformation matrix generation (shown in Figure 4.9). It obtains
the starting matrix from the schedule which brings the chosen loop vector its best
performance. For example, if this best schedule is obtained in a previous L-Search,
the starting matrix will be its default syntactic matrix, otherwise, it could be a
non-default syntactic matrix that is generated in a previous S-Search attempt.
Row/column operations are repeatedly and randomly chosen and applied until a
new syntactic matrix is generated.

Due to the "simple first" strategy, the S-Search tends to try simple syntactic
matrices before trying more arbitrary ones. The syntactic matrix can be con-
sidered as being composed of one or several portions, each of which consists of
several successive rows of the original matrix and therefore is uniquely specified
by its starting row index and the number of rows it contains. The S-Search biases
to divide the syntactic matrix evenly and into a small number of portions. In the
extreme case, each portion may contains just one row of the syntactic matrix.

The S-Search modifies each portion of the syntactic matrix with two basic operations. The first one is hoisting, which swaps values between the source and target columns of the chosen portion. For example, if the default syntactic matrix in Figure 4.16 is considered composed of only one single portion, hoisting column 5 (source) to column 4 (target) generates a new syntactic matrix as shown in Figure 4.17. Hoisting column 5 to column 2 results in a syntactic matrix as shown in Figure 4.18. The other operation is reordering, which randomly assigns new values to the target column of the portion. For example, reordering column 4 of the syntactic matrix in Figure 4.17 generates another syntactic matrix as shown in Figure 4.19.

It is worth noting that the portion(s) and both the source and target columns, as well as the new values, are parameters of the above two operations. They are randomly chosen in this default S-Search. For example, in the syntactic matrix given in Figure 4.16, the source and target columns can be arbitrarily chosen from column 0 to 5, as Figure 4.17, 4.18 and 4.19 demonstrate. Equations (4.11) to (4.15) demonstrate five of various possible divisions of a 6×6 syntactic matrix, in which rows sharing a common character belong to the same portion. Equation (4.11) considers the matrix consisting of only one portion. Equation (4.12) evenly divides the matrix into two portions. Equation (4.13) evenly divides it into three portions. It is also possible that each row is considered as a separate portion, as Equation (4.14) demonstrates, or the rows are divided unevenly among several portions, as in Equation (4.15).

\[
\begin{pmatrix}
  a & a & a & a & a \\
  a & a & a & a & a \\
  a & a & a & a & a \\
  a & a & a & a & a \\
  a & a & a & a & a \\
  a & a & a & a & a \\
\end{pmatrix}
\]  

(4.11)

\[
\begin{pmatrix}
  a & a & a & a & a \\
  a & a & a & a & a \\
  a & a & a & a & a \\
  b & b & b & b & b \\
  b & b & b & b & b \\
  b & b & b & b & b \\
\end{pmatrix}
\]  

(4.12)
// input: starting: the syntactic matrix to start with
// output: syn: new syntactic matrix

create syn as an identity matrix of starting;
repeat // when syn is illegal
{
  repeat // if syn has been generated before
  {
    randomly choose a portion p in syn;
    randomly decide an operation op and its parameter(s);
    apply op to p;
  }
}

Figure 4.20: Pseudo code of the new syntactic matrix generation

\[
\begin{pmatrix}
a & a & a & a & a & a \\
a & a & a & a & a & a \\
b & b & b & b & b & b \\
b & b & b & b & b & b \\
c & c & c & c & c & c \\
c & c & c & c & c & c \\
\end{pmatrix}
\]  
(4.13)

\[
\begin{pmatrix}
a & a & a & a & a & a \\
b & b & b & b & b & b \\
c & c & c & c & c & c \\
d & d & d & d & d & d \\
e & e & e & e & e & e \\
f & f & f & f & f & f \\
\end{pmatrix}
\]  
(4.14)

\[
\begin{pmatrix}
a & a & a & a & a & a \\
a & a & a & a & a & a \\
b & b & b & b & b & b \\
b & b & b & b & b & b \\
b & b & b & b & b & b \\
c & c & c & c & c & c \\
\end{pmatrix}
\]  
(4.15)

The S-Search gives no bias to either hoisting or reordering. It keeps applying
the operations until it generates a new and legal syntactic matrix that it has not
generated before. This procedure is expressed as pseudo code in Figure 4.20.

For efficiency concerns, the profile database stores all the legal and illegal ma-
trices that have been generated. These records are used in testing for redundancy
and legality.
// input: 1) L-pool;
LoopVector lv; // the loop vector with best performance
Schedule s, original_s;
Matrix syn, original_syn; // syntactic matrices;

assign lv whose schedule’s performance is the best in L-pool;
pick out original_s, the schedule which leads to lv’s best performance;
pick out original_syn used by original_s;

repeat // for a certain number of times
{
    generate syn, the new syntactic matrix, from original_syn;
    construct new schedule s with lv and syn;
    test s;
    store the profile into the profile database;
}

Figure 4.21: Pseudo code of the default S-Search

4.5.3 Construct the schedule

The S-Search uses the same approach as used by the L-Search to construct a
schedule from a given loop vector and a syntactic matrix. This default S-Search
is summarised in pseudo code shown in Figure 4.21.

4.5.4 Another approach to choose loop vector(s)

UTF requires that each statement in the original loop nest be given a separate
mapping. All the approaches presented above consider only the cases in which all
the mappings in a schedule share a common loop vector. Theoretically, this is not
always the case. Therefore, for coverage reasons, the heuristic search algorithm
must also consider the cases that s different loop vectors are used in mappings
of all the s statements in the loop nest. These s loop vectors are not necessarily
all different from each other. They could be chosen from the L-pool in a biased
random manner according to their performance.

In such cases, each loop vector is paired with a separate syntactic vector, or a
syntactic matrix if loop unrolling is involved, to construct the mapping. The S-
Search generates them separately by adding to the default syntactic vector/matrix
the offset of the statement in the loop nest. These syntactic vectors/matrices are
then combined with the corresponding loop vectors to generate the schedule.

This procedure of the non-default S-Search is summarised in Figure 4.22.
// input: L-pool;
Schedule schedule;
LoopVector lv;
Mapping mapping;
repeat // for a certain number of times
{
  create a new schedule schedule;
  repeat // for each statement in the original loop
  {
    choose lv from L-pool;
    pick out the syntactic vector(s) sv from lv’s schedules;
    modify sv;
    construct mapping with lv and sv;
    add mapping to schedule;
  }
  test schedule;
  store the profile into the database;
}

Figure 4.22: Pseudo code of the non-default S-Search

4.5.5 Bias in S-Search control

Because of this ”multiple loop vectors” variant of the default S-Search approach, at the start of every round of the S-Search, a decision must be made on which of these two approaches should be used, with bias to the default one.

4.6 Coverage

Coverage is one of the concerns of the heuristic search algorithm. Given no resource limit, the search algorithm should be able to visit every point in the whole search space specified, without leaving any point unchecked, although, practically, there is no need to check every point in the space.

This requirement is met with the redundancy check used in the heuristic search algorithm. In this section, the redundancy check will be explained before an informal explanation of the coverage of the search algorithm is given.

4.6.1 Redundancy check

A generic data structure, *collection*, is developed for the redundancy check. It is a set of data entries of a specific type such as a matrix, loop vectors, schedules, etc. Each data entry in a collection is unique. The collection is responsible for
storing all the data entries that have been constructed during the search process, and carrying out the redundancy test. If a new data entry is unique, it is recorded in the collection and available for retrieval, otherwise, the search algorithm will discard it.

One example of this generic collection is the transformation matrix collection. It is used in the transformation matrix generation during loop vector construction. Each newly generated matrix is checked by the collection of transformation matrices to ensure that no matrix is used more than once for the chosen pair of tile size array and unrolling factor array. Therefore, each transformation matrix is unique, and so is the loop vector constructed.

The applications of this generic collection are not global in the search algorithm. Instead, they are attached to the corresponding data structures, which results in a hierarchy of collections. On top of the hierarchy is the collection of pairs of tile size array and unrolling factor array. Each pair contains its own collection of loop vectors, which share a common tile size array and unrolling factor array but differ in their transformation matrices. Similarly, each loop vector is attached with a collection of syntactic matrices that had been used by the loop vector. Each of these syntactic matrices has a reference to the corresponding schedule. Finally, there is a collection of schedules which stores all the schedules that have been constructed and tested by the heuristic search algorithm.

All the biased decision makers used in the heuristic search algorithm periodically review all of their options by re-assigning new weights to them. Various increments and decrements are given to different decision makers according to both the collected profile and the tendency of the search. This not only biases the search to the "more promising" directions, but also helps to turn the exploration back to "less promising" directions when the "more promising" directions have been thoroughly explored. This helps the search algorithm to reach all the possibilities.

It is worth noting that as more data entries are added to the collection, it needs not only more storage, but also more time for the redundancy check when a new data entry arrives. This will increase the cost of this redundancy check. However, the heuristic search algorithm presented in this thesis is able to find good schedules before this cost gets too high, as demonstrated later in chapter 6.

### 4.6.2 Coverage of the algorithm

It is known that any arbitrary square matrix can be obtained by repetitively applying row or column transformations to a starting identity matrix. Therefore,
any arbitrary transformation matrix of a specific size can be obtained via repetitively applying the transformations specified in 4.4.2 and 4.4.5. In addition, The algorithm introduced in 4.4.2 can generate any combination of tile size array and unrolling factor array. Furthermore, the loop vector collection helps to avoid L-Search to generate a loop vector for more than once. Therefore, it is guaranteed that L-Search can generate all possible loop vectors, if given enough time and resources.

Similarly, given enough time and resources, the algorithm specified in 4.5.2 can generate any arbitrary syntactic matrix for a given loop vector, and allocate the syntactic vectors to every statement in the target code segment. The introduction of the syntactic matrix collection ensures that S-Search always tries to generate a new syntactic matrix that has not tried before, i.e. a syntactic matrix will not be considered more than once for a given loop vector. Therefore, theoretically, S-Search can generate all possible syntactic matrices for a given loop vector. Therefore, when provided a complete loop vector collection, it guarantees that, if given enough time and resources, the S-Search can cover a subspace of the target optimisation space space, which contains all the points whose schedules share a common loop vector among all the mappings.

The rest of the target optimisation space consists of points where the different loop vectors are used for mappings for different statements. With the help of the redundancy check, the random selection nature of the non-default S-Search (specified in 4.5.4) ensures that all possible loop vectors are considered for each single statement within the target code segment, so are the syntactic vectors. Therefore, all the points within the rest of the target optimisation space can be reached by this non-default S-Search, if given enough time and resources.

In brief, it is believed that the heuristic random search algorithm specified in this chapter can reach all the points in the target optimisation space specified in chapter 3, if given sufficient time and resources.

4.7 Comparison

A brief comparison is made in this section between this heuristic search algorithm and some others used in related work. This comparison focuses on the following aspects of these search algorithms: search space, i.e. the size of the optimisation space they try to explore; coverage, i.e. whether they can reach all the points within the space, if given enough resource; bias mechanism, i.e. whether they use any bias mechanism; duplicate visit, i.e. whether any point will be considered
more than once during the search process (it is worth noting that duplicate visit
does not necessarily mean that the points will be tested repeatedly, instead, they
could be skipped via redundancy check); illegal points, i.e. whether there exist
illegal points in the space; and efficiency, i.e. how quickly the search algorithm
can find good points within the space.

The optimisation space considered in [70][71] is regularly-shaped, composed of
only two parameterised transformations (loop tiling and loop unrolling) in fixed
phase order. There is no illegal point within the space. Given enough resources,
the grid-based search algorithm used in [70][71] is able to reach every point in this
space. Some points on the edge of the grid may be considered more than once
during the search process. The bias mechanism enables the search algorithm
to locate good points fairly quickly, as the experimental results demonstrate.
However, due to its drawbacks (as discussed in section 2.4.2.1), it is difficult to
apply it to a much larger and more complex optimisation space containing many
transformations, such as the one considered in this thesis.

Similar regular-shaped optimisation spaces are considered by [50], which in-
clude three transformations in loop tiling, loop unrolling and array padding. Various optimisation spaces are constructed by applying these three transformations
in different but fixed phase orders. There exist illegal points in these spaces. The
phasewise exhaustive search algorithm in [50] biases its search to the optimal
transformation found in the previous phase. It does not consider a point more
than once during the search process. Experimental results show that the random
search algorithm can quickly locate good points in the space.

More transformations are included in the spaces considered by [102]. They
include loop tiling, loop unrolling, array padding, loop distribution and loop
interchange. These transformations are arranged in a fixed phase order in order
to avoid the combinatorial explosion of the space. There are illegal points in the
space. The search algorithm presented in [102] uses a bias mechanism to locate
good points quickly, as the experimental results reveal.

In [38], the transformation sequence is constructed by randomly selecting
transformations from a pool of available transformations. As they are chosen
randomly and applied globally, even repeatedly if being picked out more than
once, the space can not be specified in an explicit and systematic manner. Legal-
ity is guaranteed due to the data-flow nature of the transformations [38] considers,
so all the points in the space are legal. However, the search algorithm may make
duplicate visits to some points in the space, if identical transformation sequences
are constructed, although the chances are small. Good transformations are given
high probability to be chosen, but [38] does not provide any evidence about its efficiency.

OSE[120] considers an optimisation space composed of various compiler options/configurations (for example optimisation level), which includes some non-iterative reordering transformations not considered in this thesis. However, many points in its optimisation space have already been pruned by the compiler writer’s prior knowledge before the space exploration starts. This indicates that OSE can not guarantee space coverage. The breadth first tree search strategy OSE uses is very similar to the "simple first" and "explore to depth if a good hint is found" strategies used by the heuristic random search algorithm, which are specified in 4.2.6 and 4.2.7. Experimental results show that OSE can yield significant performance improvement.

UTF is used to specify the optimisation space explored by [67] and [99][100]. The optimisation space is much larger than those discussed above because of the following reasons. First, UTF can represent all the iteration reordering transformations and their arbitrary combinations. On the other hand, no fixed phase order is imposed because of the uniform way in which UTF represents the arbitrary combinations of these transformations. Because the search algorithm used in [67] considers the search as a mapping construction process in a level by level manner, no illegal points will be considered, and duplicate visits to a point are also avoided. The mapping construction is biased to the "sufficiently good" direction according to the estimate it made on the partly constructed mapping, but neither the space coverage nor the efficiency of the algorithm is provided. A genetic algorithm is used by [99][100] to explore this optimisation space. Although a bias mechanism is included, it can not prevent the search algorithm from both considering illegal points and making duplicate visits to the points in the search space. This GA-based algorithm is very poor in efficiency, and the chance is very small of it reaching all the points in the space.

The heuristic random search algorithm presented in this chapter considers approximately the same optimisation space as that which is explored by [67] and [99][100]. The only difference is that loop coalescing is not included in the space. As discussed above, with the help of some auxiliary data structures, the heuristic random search algorithm is able to cover the whole search space if given enough resources. It excludes illegal points via the legality tests provided by UTF, and avoids duplicate visits by keeping a record of every point it has reached during the search process. The search is biased to both the simple schedules and the variants of those good schedules found in its previous attempts. This helps the
search algorithm to achieve good efficiency, as demonstrated later in Chapter 6.

In brief, this heuristic random search algorithm tries to explore an optimisation space which is much larger but still tractable compared to most of the spaces the previous work focuses on. The heuristic random search algorithm is capable of finding good points quickly.

4.8 Summary

This chapter has presented a heuristic search algorithm for a compiler to explore the target search space in search for best optimisation schedules. The search is considered as a two phase process in which the L-Search and S-Search are carried out in roughly alternating manner. They have been presented in depth, followed by a brief and informal discussion of their coverage. A comparison has been made at the end of this chapter between this search algorithm and other algorithms used in some related projects.
Chapter 5

Learning Optimisation

5.1 Introduction

This chapter describes an optimising compiler that is enhanced by machine learning techniques. It can learn from its previous program executions and apply the knowledge gained in order to achieve performance improvement for new programs.

Section 5.2 introduces the concept of a learning compiler. It gives a summary of the system requirements and provides a review of some existing machine learning approaches. Section 5.3 explains in depth the learning optimisation approach developed in this thesis, before section 5.4 makes a comparison between this approach and others in related work.

5.2 Learning Compiler

This section introduces the concept of a learning compiler. It discusses what a learning compiler is expected to achieve, before giving a review of some existing machine learning approaches that inspire the learning approach used in this thesis.

5.2.1 Concept

An iterative optimising compiler collects useful information during its search process. For example, in this thesis, this information appears in the form of a transformation schedule and performance improvement pair. However, this information is program- and environment-specific and therefore not generally applicable. In addition, it may only reveal the characteristics of a very small number of points in the huge optimisation space which the compiler is to explore in pursuit of performance improvement.

Despite these disadvantages, this information is still useful, in that it can help to characterise or model the optimisation space, so that if asked to explore the

93
space to a greater depth, the compiler can develop useful transformations from it to locate even better points. Furthermore, when faced with a new environment, the compiler could use the previous experience from other environments to develop transformations in order to increase the chance of finding good points within a limited number of attempts. In addition, knowledge of poor prior optimisations contains useful information as well, in that it can provide information that certain points, subspaces or transformations are of little use and therefore need not be explored further. In brief, all the information collected from previous optimisation attempts are useful, regardless of being positive or negative, or their context.

A learning compiler is defined as a compiler having the ability to acquire knowledge from its previous efforts and to apply this knowledge when needed to further explore the search space, to compile another program, or even to compile another program in another environment. A learning compiler needs other auxiliary abilities to fulfill its task efficiently, for example, information storage and retrieval.

A learning compiler can be thought of as an iterative optimising compiler with machine learning enhancements. It successively applies different transformations to the program, evaluates their performance, and collects the profile information during the course of the program execution, as an iterative optimising compiler does. However, it is different in that it uses machine learning techniques to retrieve prior knowledge from not only the profile of the current program, but also the profile of previous programs it has met. This prior knowledge is retained by the compiler and will be used later, as suggested before.

In recent years, significant progress has been made of machine learning[94], with successful applications in an increasing range of areas from data-mining to autonomous vehicle control. Despite this, there have been few attempts in introducing machine learning technologies into compiler design, as summarised in the Related Work chapter.

A typical approach[94] to develop a machine learning system must consider the selection of training examples, target function and its representation, as well as the function approximation algorithm.

### 5.2.2 System Objectives

Discussed below are some key system requirements of a learning compiler, together with some justification of decisions required during the system design phase.

First of all, effectiveness is the dominant concern for a learning compiler.
The approach is expected to provide useful transformations, if there exists any improvement. Ideally, it should deliver most of the potential performance improvement for the given program. Considering the large optimisation space to explore and the unknown distribution of good points within it, effectiveness is the dominant concern for a learning compiler.

In many circumstances, a transformation is expected to be provided within a specified time. For instance, it is not acceptable that such an approach take excessive amount of time before providing a transformation which gives only a negligible speedup to a simple program. It is reasonable to expect the learning approach to respond within time constraints.

Certain machine learning approaches require more time than others for in-depth learning/analysis. Such costs could be paid off by transformations that achieve a greater performance improvement. Thus, there is a tradeoff to be made between the effort spent and the performance gained.

It is noticed that some existing machine learning approaches are expressed in a formal and abstract manner, which usually takes more effort to integrate into existing systems than the others. Due to the demand for timely response, it is not acceptable for a learning compiler to spend an excessive amount of time on the system integration and result interpretation. Therefore, it is desirable that the learning scheme could be expressed in an explicit and applicable manner, so that it can later be easily integrated and interpreted.

Finally, there is a requirement for portability. Clearly, a general approach independent of language, compiler and architecture is more preferable than an environment-specific one. This environment independence will save effort in adapting to other compiler systems when necessary.

### 5.2.3 Existing machine learning approaches

Machine learning is a subject that draws on many other subjects such as statistics, artificial intelligence, information science, biology, cognitive science and philosophy. Due to this multi-disciplinary nature, many machine learning approaches appear similarly applicable for a learning compiler at first glance. This subsection reviews some of the existing machine learning approaches which have been shown effective in practice, and discusses their applicability for a learning compiler.

#### 5.2.3.1 Artificial neural networks

Artificial neural networks (ANN)\cite{29}\cite{94} provide a general, practical and robust approach to approximate a real- or discrete-valued target function from training
examples. It has been shown to be among the most effective approaches currently known[94]. Inspired by biological neural system, ANN is built out of a densely-interconnected set of simple units called perceptrons, each of which is a simple processing unit taking a number of inputs (possibly the outputs of other units) and producing a single output. Its problem-solving ability is based on the ability of the individual perceptrons as well as their interconnection. ANNs are suitable for problems where the training data correspond to noisy, complex sensor data in form of attribute-value pairs, long training time is acceptable, and fast evaluation of the target function is required. The backpropagation algorithm is a well-known and widely-used method to train ANNs.

Considering the task the learning compiler faces, the training examples may contain not only transformation attributes, but also program and environment features and other factors, which means that there may be a variable rather than fixed number of system inputs. Similarly, as the learning compiler is expected to provide transformations when new problem arises, the number of system output is expected to be variable rather than fixed, but ANN lacks this flexibility.

In addition, ANN usually expects both its architecture (including the number of layers of perceptrons, their interconnections and system input and output) and the target function to be decided before the learning process starts, or at least be relatively stable even if change is feasible during the learning process. The lack of runtime adaptability makes it difficult for ANN to meet the above requirements of a learning compiler.

Finally, the ANN’s requirement of long training time may be too expensive for a learning compiler in many circumstances.

5.2.3.2 Genetic algorithm

Genetic algorithms (GA)[40][94] provide another approach aimed at finding the optimal hypothesis according to a predefined fitness function. It is based on an analogy of biological evolution in that it iteratively seeks better solutions to problems in the same way that evolution optimises the population of living organisms for their habitats and environments. It considers an organism in the population as an abstract representation of a potential solution in a problem domain, associated with it a corresponding fitness value. Therefore a population of organisms represents a subset of possible solutions. Solutions of higher fitness values are chosen more frequently for reproduction (in the form of operations such as mutation and crossover) than those of lower fitness values.

GAs have been widely used to solve optimisation problems. For example,
see the GA-based projects reviewed in chapter 2. Although [99] claims that GA optimisation can produce code with higher performance than that produced using SGI's state-of-the-art commercial compiler, the disadvantage of GA is still obvious, it is destined to achieve its target very slowly due to its random selection nature. GA has very limited ability to analyse and remember what makes a certain organism better than others. Instead, it relies on random selection. Therefore, the majority of organisms it creates via reproduction are of little use to achieve the final target. The effort spent to create and test them is wasted.

5.2.3.3 Instance-based learning

Instance-based learning (IBL)[94] is considered as a straightforward approach to approximate a target function. It simply stores all the training data as instances during its learning process. When a new query instance is encountered, it retrieves a set of similar instances and uses them to classify this new instance. This "lazy" approach saves IBL the effort of constructing an explicit hypothesis of the target function over the entire instance space. Instead, it constructs a local approximation to the target function for each query instance, which could be more precise than an approximation performing well over the entire instance space. For each query instance, the approximation of the target function could be very different.

The advantage of IBL is its ability to model a complicated target function by a collection of less complicated local approximations. However, this ability comes with the high cost of classification of a new query instance. Additionally, because IBL usually considers all of the attributes during classification, the distance between two instances may be far if the target concept depends on only a few of many available attributes. This is another drawback of IBL.

Similar to ANN, the variation in the number of attributes used as system input is a big concern when a compiler considers using IBL techniques. It is difficult for IBL to present all the potential transformations in a way that a compiler can understand and use without much difficulty.

Case-based reasoning (CBR)[73][94] is considered as a variant of IBL, in which instances (cases) are represented by richer symbolic descriptions than the real-valued points in a Euclidean space used in IBL. This makes a more accurate expression of the problem, the training data and the query feasible. In many cases, CBR relies on knowledge-based reasoning[94] and search-intensive problem-solving methods to find the proper combination from multiple cases, in order to provide the solution to a new query. However, due to the current technology, it is
difficult for a compiler to implement knowledge-based reasoning without human intervention.

5.2.4 Decision

Besides ANN, GA and IBL, there are many other machine learning approaches[94], such as decision tree learning, Bayesian learning, analytical learning, hypothesis evaluation, general-to-specific ordering for concept learning, etc. However, they have different application domains, which are not directly applicable for a compiler. Therefore, they are not considered suitable for a learning compiler, considering the latter’s objectives in effectiveness, timeliness and applicability.

As for these three applicable approaches, the long time needed to train an ANN and to achieve good results in GA makes it difficult for a compiler to adopt these two approaches. It is believed that IBL is a reasonable strategy for a learning compiler. There have been some projects[86][87] in applying CBR in a specific optimisation problem, as reviewed in the Related Work chapter. Their results have demonstrated the feasibility and potential of IBL in this area.

5.3 Instance-based Learning Optimisation (IBLO)

5.3.1 Basic ideas

The basic idea is that a learning compiler makes a decision on how to optimise a particular program based on its previous experience. Each time a program is optimised, the transformation is applied and the resulting performance are stored along with a description of the program. On encountering a new program to optimise, the compiler searches the database of previous cases for similar programs. Transformations that were beneficial to similar programs are then considered for the current program.

In order to achieve this, the learning compiler must correlate programs, the transformations applied and the resulting performance improvement in a systematic manner. In machine learning terms, the inputs or features of the problem are the transformation(s) and a description of the program and environment, with the output being either a transformation or execution time. These problem features not only reveal the important details of the program and environment, but also help the compiler to classify for later retrieval, as demonstrated later in this chapter. They have to be formally specified in order to allow the application of instance-based learning.
It is known that transformations can bring performance improvement only when applied under certain conditions, i.e. when either the target program or the environment, or both, demonstrate certain features. In other cases, they may degrade rather than improve performance. For example, loop skewing can bring performance improvement to wavefront computations only. Loop fusion requires that the two loops to be fused must have the same loop bounds, and there is not such a dependency $S_2 \Rightarrow S_1$ in the fused loop, where $S_1$ and $S_2$ are from the first and second loop respectively. Similar legality restrictions of various transformations can be found in [17].

A transformation's applicability and potential benefit depend on not only the transformation itself, but also the features which characterise the given program and environment, such as those given in section 3.5 in order to extend the UTF representation in this thesis. This observation also stands for transformation sequences.

Therefore, a learning optimisation approach must extract from training examples not only how much benefit a transformation can bring, but also the features of the program and environment on which it is applied. In brief, it can be said that the performance gain $G$ is a function $func$ of the program $P$, the transformation(s) $T$ and the environment features $F$, where $P$, $T$ and $F$ may each contain many different factors.

$$G = func(P, T, F) \quad (5.1)$$

Therefore, the kernel of the learning compiler is considered as a multi-dimensional analysis on knowledge of all these factors. It has to explore the underlying relations among program features, environment features, transformations and performance gain in order to approximate the above function.

The main tasks of instance-based learning optimisation (IBLO) are classification, knowledge storage and transformation selection. Classification places each training examples into different categories according to their program and environment feature values. Knowledge storage stores and maintains a list of schedules that bring performance improvement for each category. Transformation selection means that, when a new program arises, IBLO must try to select an appropriate transformation or transformation sequence and apply it to the program. The result may also be used as a new training case. These three tasks are explained in depth in the following subsections.

It is worth noting that this approach does not necessarily need a separate learning phase, nor a set of training examples given in advance. It initially starts
its experience accumulation process with its first attempt to optimise a given program using the heuristic random search approach presented in the previous chapter. It learns on the fly and iteratively applies what it has learned to new programs, taking each attempt as a new training example. Previous optimisation efforts on other programs and/or other platforms may also be used as training examples.

5.3.2 Classification

Given a set of program and environment features and a set of transformations, one way to find out the underlying relations among them is to classify the programs into different categories and see how different transformations perform on programs of each category, especially, which transformations bring most significant performance improvement to these programs.

When a new program is encountered, it is first categorised by the classification algorithm, and then the corresponding transformation or transformation sequence is applied. This is based on the observation that programs with similar characteristics usually benefit from the same transformation(s)\cite{80}\cite{86}.

Program and environment features usually appear in various forms (boolean, integer, floating points, etc.). For example, for the features listed in section 3.5, values of feature 2a (loop nest depth) are integers, whilst those of 2c (whether the loop is a perfectly nested loop(s)) and 2e (whether it contains abnormal exit)) are booleans. In addition, these features may be valid within different boundaries. For example, loop nest depth usually varies only within a small range such as [1..5] in most cases, whilst loop sizes may vary from 0 to \infty. A pre-processing is needed to transform the raw training data into a unified or similar form for IBLO to analyse, store and retrieve in an easier manner. Furthermore, it is understandable that not all the program and environment features play an equal role in performance, so different weights should be assigned to different features during the classification. The weights used for different features can be drawn from previous experience and later re-adjusted during the learning process, similar to the heuristic search algorithm in this thesis.

Classification can become biased or over-fitting\cite{94} in some cases, depending on not only the training examples themselves, but also the order in which the system receives them. The classification algorithm must therefore adjust itself to the training examples.

All possible combinations of program and environment feature values are considered equal to allow easy classification. Each of these combinations is regarded
as a separate category and assigned a separate entry when the compiler meets a program in this category.

This classification in Cartesian product manner is naive, as the number of features and the valid values of each feature are both large, which inevitably leads to a very large number of entries, making this scheme inefficient when classifying a new program or locating one specific entry for a new query instance. Nevertheless, because entries are created and stored only if such categories do appear in training examples, and they are sparsely distributed, the number of such non-empty entries does not become too large to handle.

Several definitions must be given before the classification algorithm for a new query can be introduced.

The similarity of a given program $P$ to a given category $C$ is defined as in Equation (5.2).

$$similarity(P, C) = \sum_{i=0}^{n} match(f_i(P), f_i(C))$$  

(5.2)

where

$$match(a, b) = \begin{cases} 
1 & \text{if } a = b \\
0 & \text{otherwise}
\end{cases}$$

(5.3)

and $n+1$ is the number of program and environment features, such as those listed in section 3.5, $f_i(P)$ and $f_i(C)$ are the values of feature $f_i$ for $P$ and $C$ respectively.

If, for the given $P$ and $C$, $similarity(P,C)$ equals the number of features, i.e. $n+1$, then it is claimed that $P$ belongs to category $C$, otherwise, $P$ is claimed similar to $C$. It is understandable that the higher the value of $similarity(P,C)$, the more similar $P$ is to $C$, and vice versa.

For example, consider a small set of four features \{ $f_0, f_1, f_2, f_3$ \} and a category $C$ which is characterised by feature value vector $(2,2,2,3)$. For a program $P_0$ whose feature value vector is $(2,2,2,3)$, $similarity(P_0, C)=1+1+1+1=4$, which is equal to the number of features, shows that $P_0$ belongs to $C$. For another program $P_1$ which is characterised by a feature value vector $(2,2,1,0)$, $similarity(P_1, C)=1+1+0+0=2$, which is not equal to the number of features, shows that it is just similar to $C$.

Considering the fact that different program and environment features may contribute differently to the program performance, a more general definition of similarity must assign different weights to different features, as shown below in Equation (5.4). It is noted that Equation (5.2) is a simplified/special case of (5.4), in that all the features are assigned equal weight of 1.
\[
\text{similarity}(P, C) = \sum_{i=0}^{n} (w_i \times \text{match}(f_i(P), f_i(C)))
\]  (5.4)

Accordingly, it is claimed that program \( P \) \textit{belongs to} a category \( C \) if \textit{similarity}(\( P, C \)) equals the sum of feature weights, i.e. \( \sum_{i=0}^{n} w_i \), otherwise, \( P \) is claimed \textit{similar to} \( C \).

Similarly, the \textit{program similarity} and \textit{category similarity} are defined as in Equation (5.5) and (5.6) respectively, where \( P_1 \) an \( P_2 \) are two programs and \( C_1 \) and \( C_2 \) are two categories.

\[
\text{similarity}(P_1, P_2) = \sum_{i=0}^{n} (w_i \times \text{match}(f_i(P_1), f_i(P_2)))
\]  (5.5)

\[
\text{similarity}(C_1, C_2) = \sum_{i=0}^{n} (w_i \times \text{match}(f_i(C_1), f_i(C_2)))
\]  (5.6)

The classification algorithm keeps all of the existing category entries in a list. When a new entry \( P \) arises, comparisons are made between it and all the category Cs in the list on \textit{similarity}(\( P, C \)). The results are kept in a list in descending order of their similarity values. Therefore, when the comparisons are finished, \( P \) either belongs to the category on the top of the list, or is most similar to the top few categories. This classification algorithm is inefficient if there are a large number of categories in the list, due to its sequential search manner.

It is worth noting that this classification algorithm can be replaced by other algorithms if needed. For example, a decision tree can be generated from the existing categories and then used to classify new queries. Compared to the above sequential search algorithm, this tree-based classification algorithm can locate the exact category for the new entry much faster, if the category does exist. However, if the exact category has not yet been created, IBLO needs to consider categories similar to the new query with respect to different features. These categories reside on different branches of the tree, making it difficult for the tree-based classification algorithm to locate all of them. Additionally, in order to adapt the tree to the categories encountered later and to maintain its efficiency, the tree must be re-structured periodically. Considering the number of potential categories, the frequency of potential weight readjustment, the difficulty of locating multiple similar categories for a new query, and the effort needed for tree re-structuring, it is believed that the decision tree algorithm is not a very good alternative.

Although the sequential classification algorithm is less efficient, it provides stable behaviour with regard to all but the first factors mentioned above.

102
5.3.3 Knowledge storage

For each category, IBLO stores all the schedules that have been tried. These schedules may vary in depth, loop vectors used, syntactic matrices, parameters and other factors. Intuitively, it means that there may exist different transformations and transformation sequences that can bring performance improvement to programs of a category. For example, as shown in Figure 3.1 and 3.2, both loop tiling and loop unrolling can improve the performance of program C, with many different tile sizes and unrolling factors.

For efficiency, a category will not be constructed and stored until the compiler has obtained experience from programs of it. This means that the IBLO will not store an empty category with no transformation included.

It is worth noting that the maximum number of categories depends on the number of program and environment features and their values, rather than the number of programs the compiler encounters. When the number of programs encountered increases, the compiler needs more space for knowledge storage, as well as more time for knowledge retrieval. This will result in slow response to new optimisation request. One possible solution is inspired by the idea of indexing. The compiler may store only a model it builds from the transformations collected, and store the original ones in a backup storage which is accessed only when necessary. This model can be considered as a concise summary of all the transformations, from which one may be drawn when new request arrives. It must be updated periodically to include new transformations collected since the last update. A simpler alternative is to store not all but only a reasonable number of best transformations in each category, and discard all the rest. However, it may result in losing useful information, which will affect the quality of transformations chosen for new programs. This is beyond the scope of this thesis.

Since the schedules can be interpreted as transformation sequence, as shown in the previous chapter, for each category the schedules are grouped into areas according to the type of transformation(s) they use, with each area storing information about the use of a specific transformation or transformation sequence, i.e. the parameters used. For example, one area is created for schedules of loop tiling, one for the schedules of loop unrolling, and one for the schedules of tiling and unrolling. Each point in the area stands for an application of this transformation in one training example of this category, along with a set of transformation parameters.
5.3.4 Transformation selection

When a new program \( P \) arises, IBLO must first classify it, and then select transformations from the schedules from the category \( P \) belongs to or from some categories it is most similar to.

IBLO uses the same classification algorithm presented in 5.3.2 to find the category which \( P \) belongs to. If there exists such a category \( C_0 \), the transformation(s) for \( P \) will be selected directly from the compiler’s prior experience with \( C_0 \). Otherwise, IBLO will find one most similar category \( C_1 \) and use its prior experience on \( P \). If there is more than one category that \( P \) is most similar to but differ with respect to various features, the transformation(s) could be selected from the experiences from all these categories, using a density-biased random approach to be explained below. Currently, the top three most similar categories are considered in such cases.

Problems may arise when there exist only a few similar categories. Theoretically, even in such cases, the classification algorithm can still find a category \( C \) which is more similar to \( P \) than the others are. In practice, \( C \) may not be similar enough to \( P \) to provide applicable and good transformation(s), in which case, the random heuristic search algorithm presented in chapter 4 could be used as a backup. A threshold can be set to decide whether to select transformation(s) from \( C \) or to try random search instead. It is understandable that this threshold is relevant to the program and environment features and their contributions to the classification. This problem has not been addressed in this thesis, and will be explored in future work.

A variant of this approach is that IBLO may use experience not only from the exact or most similar category, but also from other very similar categories, for example, the first few entries on top of the category list in descending order of their similarities to \( P \).

Once the category is found, a simple algorithm is used to select transformation(s) from the available experience.

For a given category, the possibility that a transformation or transformation sequence can bring performance improvement to its programs can be considered as a function defined on the optimisation space. According to the \( k \)-nearest neighbour algorithm[94], for any point in this space, its possibility of improving program performance is decided by the performance of its neighbouring points. If the majority of them can bring performance improvement, it is likely that the transformation(s) this point represents can also bring performance improvement. Therefore, it is believed that the more crowded an area a point is in, the more likely
the transformation(s) it stands for can improve the performance of a program in this category. Points are grouped into areas according to the transformations they represent, for instance, one area of loop tiling, one area of loop unrolling, one area for both loop tiling and unrolling, etc.. Furthermore, areas can be divided into sub-areas according to the parameters their points use, as demonstrated later in this section.

Because the transformations vary in terms of the performance improvement they can bring, a modification is made to the above assumption. It becomes the more significant an area a point is in, the more likely the transformation(s) it stands for can improve the performance of a program in this category. The significance of a given area \( A \) is defined as below:

\[
\text{significance}(A) = \sum_{i=0}^{n} sf(s_i) \tag{5.7}
\]

where \( A \) consists of a set of points \( p_0, p_1, \ldots, p_n \), whose performance improvement are \( s_0, s_1, \ldots, s_n \) respectively, and \( sf \) is a function defined on \( s_i \). A simplified definition may appear as

\[
\text{significance}(A) = \sum_{i=0}^{n} s_i \tag{5.8}
\]

which is defined on \( s_i \) directly.

In the cases where more than one category is chosen for the given program, a virtual category is constructed whose areas are obtained by combining the corresponding areas of these categories. For example, if, for the given program \( P \), IBLO finds two most similar categories \( c_0 \) and \( c_1 \), \( c_0 \) contains two areas \( a_0 \) and \( a_1 \), and \( c_1 \) contains three areas \( a_1, a_2 \) and \( a_3 \), then a virtual category will be constructed for \( P \) with areas \( a_0 = c_0.a_0, a_1 = c_0.a_1 \cup c_1.a_1, \) and \( a_2 = c_1.a_2 \).

IBLO will select transformation(s) from these areas, using a selection algorithm summarised in Figure 5.1.

In the iterative division process, area \( A \) is divided evenly into a number of sub-areas in order to find the most significant sub-area. For example, a 2-dimensional area is divided into 4 (2×2) or 9 (3×3) sub-areas. Practically, IBLO may consider only a certain number of points on top of the performance improvement list and neglect all the rest. Therefore, the most significant area can be located within a reasonable number of rounds.

A problem that may arise is that the most significant area may overlap two or more sub-areas rather than be within one sub-area in the previous iteration. Consider that the target is to locate the most significant area rather than to
// input: category C
// output: a transformation or transformation sequence

calculate the significance of each area of the category C;
choose the most significant area, namely A;

repeat // if A contains too many points
{
    choose a plan to divide A evenly;
    divide A into sub-areas according to the chosen plan;
    calculate the significance of each of these sub-areas;
    let A be the most significant sub-area;
}

// area A now contains only a reasonably small number of points
randomly pick a point h from A;
return the schedule h standing for;

Figure 5.1: Pseudo code of the transformation selection

precisely specify it, even if overlap occurs, part of the most significant area is still
within one of the overlapped sub-areas and makes it significant. This sub-area
will then be chosen for further division, which will not affect the final result. In
addition, previous experience[70][74][80][102] shows that it is rare that only a few
points outperform all other good points in the area. Instead, there are usually
many points which achieve similar performance improvement. So the probability
of picking a good point from many will not be too low. Therefore, it is believed
that this overlap problem may have an impact on the result but it is likely to be
insignificant.

It is worth noting that this algorithm does not keep dividing the area A
until it contains only one point to be used as the final output transformation or
transformation sequence. Instead, the division process stops once A contains just
a small number of points, for instance five or six in this thesis. This is because
the possibility can never be eliminated that the classification algorithm neglects
some difference between two programs, and neither can the possibility that this
neglected difference may cause a big difference in runtime behaviour. Therefore,
no matter how similar two programs are, it is never guaranteed that any one good
point for one program, even if it is the best one, is applicable for the other and
able to bring performance improvement to it. For example, loop distribution may
bring good performance improvement to a program $P_0$, but is not applicable for
another program $P_1$ which is considered most similar to $P_0$ by IBLO, due to $P_1$'s
legality constraints.

Therefore, in the final step, the final transformation or transformation sequence is selected randomly from area $A$ which contains a reasonably small number of points, rather than always selecting the best point from it. It is expected that such an approach can increase the probability of picking a good and applicable transformation or transformation sequence on average. A variant of this approach is to bias the random selection with the points’ performance improvement significance. Its efficiency remains to be evaluated.

Furthermore, even with this random selection algorithm, it is still possible that the chosen transformation or transformation sequence is not applicable for the target program. If it is caused by parameter missing (for example, a transformation for a single loop is chosen for a double nested loop), IBLO will choose random value(s) for the missing parameter(s) from a proper range. If it is caused by other reasons (such as legality), it simply tries to choose another point from $A$, or even from another area $A'$ if necessary. This process repeats until an applicable transformation or transformation sequence is selected.

Figure 5.2 demonstrates how this transformation selection algorithm works. Suppose the algorithm has located a category whose transformation area is a two-dimensional space as shown on the left, where each point stands for a transformation of equal significance. The algorithm divides this area evenly into four subareas $A_0$, $A_1$, $A_2$ and $A_3$ which have 8, 3, 2 and 1 transformation each, as indicated. Because $A_0$ is the most significant subarea, it is chosen for further division, which results in the figure on the right hand side of Figure 5.2. The resulting subspaces $A_{00}$, $A_{01}$, $A_{02}$ and $A_{03}$ contain 2, 1, 2 and 3 transformations respectively. No more division is needed as each subspace now contains only a small number of points. Because $A_{03}$ is the most significant of all, the algorithm will randomly pick one from its three points, and uses the transformation it represents as the final result.

5.4 Comparison

The learning optimisation approach presented above is based on instance-based learning. It considers the search problem as a regression of the function between programs and transformations, where high performance improvement is expected to be achieved on the regression made for the new query instance. In order to find the proper category (or categories) for the new program, it uses a classification algorithm which shares the same spirit with the $k$-nearest neighbour algorithm[94].
After the classification, IBLO still has to draw proper transformation(s) from the category (or categories), which is beyond the consideration of instance-based learning.

Case-based reasoning is a promising variant of instance-based learning. It represents both training cases and new queries as complex logical descriptions rather than points in Euclidean space as used in instance-based learning. However, case-based reasoning relies on knowledge-based reasoning and search-intensive problem solving methods to retrieve and combine cases for the new query case. These techniques are not considered in the learning optimisation approach presented above, due to their high cost and need for human intervention.

The ability of case-based reasoning to solve optimisation problems has been demonstrated in [86], which presents an interactive tool capable of providing performance tuning guidance to the user. It checks a list of program features and, if certain conditions are met, suggests the user try specific transformation(s). It has not addressed the problem of parameter selection, which has significant impact on whether and how much the transformation(s) can affect the performance, as shown in Figures 3.1 and 3.2 in chapter 3 and some other iterative optimisation projects[50][70][71][102]. The parameter selection problem is addressed in [87], which considers only one transformation in loop unrolling, as discussed in chapter 2. Another weakness of [86] is that it does not consider the problem of transformation combinations, which can provide performance improvements beyond that of single transformations, as demonstrated in [38][50][70][71][80] and other related work.

Similar to [86], IBLO checks a list of program and environment features in order to identify optimisation opportunities. It is able to provide more concrete transformations with parameters specified than [86]. In addition, transformation
combination is considered. Furthermore, if iterative optimisation is allowed, the results of applying these transformations can be used as further training examples for the learning compiler, which, in turn, will improve the ability of the learning compiler. It is believed that integrating both IBLO and the heuristic random search algorithm can further improve a compiler’s performance.

5.5 Summary

This chapter has presented an instance-based learning optimisation approach. Section 5.2 has introduced the concept of a learning compiler, before giving a summary of its system requirements and providing a review of some existing machine learning approaches. Section 5.3 has explained in depth the instance-based learning optimisation approach. A brief comparison has been made between this learning approach and some related work.
Chapter 6

Experimental Results

6.1 Introduction

This chapter introduces the implementation of the heuristic search algorithm and the instance-based learning optimisation approach (IBLO), before presenting the experimental results. The prototype AOF-Java is introduced in section 6.2, then the experimental setup and benchmarks used in the experiments are specified in sections 6.3 and 6.4 respectively. The experimental results of the heuristic search algorithm are presented and discussed in section 6.5, before those of IBLO are given and analysed in section 6.6. Section 6.7 gives the experimental results of IBLO on a large benchmark, followed by a brief summary at the end of the chapter.

6.2 AOF-Java

Adaptive Optimisation Framework for Java (AOF-Java) implements source-to-source Java transformation. It aims at providing continuous performance enhancement to Java applications via feedback-directed iterative optimisation and machine-learning techniques based on the mapping notation presented in Unified Transformation Framework. The prototype AOF-Java is implemented in Java, with the help of JavaCC[60] for the compiler implementation.

6.2.1 Architecture

AOF-Java has a distributed architecture similar to ADAPT[121][122][123]. The client executes and profiles the program, whilst the server is responsible for program analysis, transformation, compilation, offline profile analysis and making transformation decisions. It consists of three major components: the Search Engine, the Compiler and the Profiler, as shown in Figure 6.1.
This distributed architecture and workload allocation enable profiling with minimum interference. In the prototype AOF-Java, the server and client reside on different computers sharing a common file server, and they communicate with each other via a Java Remote Method Invocation (RMI) interface[44].

6.2.2 Search Engine

The Search Engine implements the heuristic search algorithm described in chapter 4. It can easily be adapted to other search algorithms, or be directed from an external source, such as a learning approach.

In the prototype AOF-Java, the Search Engine chooses to interpret the schedule as a transformation sequence as most traditional optimising compilers do, rather than using the equivalent "one-stop" code generation algorithm provided by UTF. This is because this algorithm is inefficient and does not provide a Java interface. The advantage of this traditional approach is that, without too much effort, the compiler’s ability can be extended by adopting other non-iteration-reordering transformations such as method inlining and scalar replacement of array elements[96] etc.. Thus we have the coverage of the UTF-based space without being restricted to it. In addition, this approach makes it feasible for other
optimising compilers to adopt this search algorithm if proper interfaces are provided.

Each attempt of the L-Search generates a schedule. The tile size selection generates a specific loop tiling, and the unrolling factor selection generates a specific loop unrolling. As explained in the Heuristic Search chapter, the transformation matrix can be considered as derived from a default transformation matrix in a step by step manner, each step of which is equivalent to a specific transformation. It is therefore considered equivalent to a sequence of transformations which is constructed during the course of transform matrix generation. This transformation sequence is then attached to that generated in the previous two optional steps, if any, to construct the complete transformation sequence which represents the schedule generated by the L-Search.

Because the syntactic matrix is generated in a manner similar to the transformation matrix generation in L-Search, it is interpreted in a similar manner. The resulting transformation sequence of the S-Search consists of transformations such as loop fusion, distribution and statement reordering. It is attached to the end of the transformation sequence of the loop vector to construct the final transformation sequence.

The Search Engine uses a domain-specific language to specify the transformation sequence. The result is stored in a .decision file, which will be processed by the AOF-Java Compiler.

The Search Engine records its search process in a .log file, which consists of transformations and the corresponding profile.

### 6.2.3 Legality test

The prototype AOF-Java currently provides semi-automatic legality test facilities except legality constraint detection. The user uses a domain-specific language to specify the legality constraints (at statement level) of the given program in a separate .dependency file.

When launched, the Search Engine reads all the legality constraints from the .dependency file. It carries out the legality test for each of the schedules it generates during the search process. If the new schedule passes the test, the Search Engine will send the corresponding transformation sequence to the Compiler for compilation, before the latter sending it to the AOF-Java Client for evaluation. Otherwise, the schedule is abandoned, and the decisions made for it are reviewed, which may lead to re-adjustment of the bias weights. It is worth noting that all schedules will be stored in the profile database, regardless of their legality test.
results.

### 6.2.4 Compiler

The AOF-Java Compiler consists of three major components: the Analyser, the Transformer and the Unparser.

The Analyser reads the input Java program, collects useful program information, and constructs the syntax tree. The Transformer then modifies the syntax tree according to the transformation sequence it receives from the Search Engine. It decomposes the transformation sequence and dispatches the transformations to the corresponding syntax tree rewriters. Each rewriter carries out a specific transformation by modifying the syntax tree according to the parameter(s) received.

Currently, the prototype AOF-Java explore the UTF-based optimisation space with the following syntax tree rewriters of iteration reordering transformations: LoopAligner, LoopDistributor, LoopInterchanger, LoopReverser, LoopTiler and LoopUnroller. It also implements other non-iteration-reordering transformations such as method inlining and scalar replacement of array elements[96]. In addition, the Transformer is also responsible for inserting profiler stubs to the program for profiling purpose.

Once all the transformations have been carried out, the Unparser parses the modified syntax tree and outputs the resulting Java source for compilation. The AOF-Java Compiler uses javac as the bytecode generator. It uses AOFjavacLauncher to launch the compilation. No native code generator is included in the prototype AOF-Java.

### 6.2.5 Profiler

During program execution, runtime information is collected by the profiler stubs which are inserted into the program by the Compiler. The prototype AOF-Java profiles only for-loops as it focuses on iteration reordering transformations. An example of a loop under profiling is shown in Figure 6.2.

The light-weighted profiler stub uses a timer to evaluate the execution time and gives the results in milliseconds. At the end of the execution, it outputs the profile to a .profile file. It is worth noting that other performance measures may be used in the profiler stub.

The AOF-Java Profiler currently provides online profiling and offline profile analysis. Once the program execution is over, The ProfileAnalyser reads the
ProfilerStub.addTimer("L","""");
ProfilerStub.startTimer("L");
L: for (int i=1; i<N; i++) {
    x[i] = x[i-1] +y[i];
}
ProfilerStub.stopTimer("L");
...
ProfilerStub.output("benchmark.profile");

Figure 6.2: Code segment with ProfilerStub

.profile file and saves the collected profile to the profile database. The profile is also sent to the Search Engine to direct the search in the optimisation space.

6.2.6 AOF-Java Client

The AOF-Java client is responsible for program execution and profiling. When it receives a program to compile, it sends a compilation request to the server and then waits for the response. On receiving the "compilation finish" notification from the server, it uses AOFjavaLauncher to launch the execution via java. By default, java launches a JIT compiler to turn Java bytecode into executable before the execution starts. Once the execution is finished, it notifies the server to start profile analysis and prepare the next schedule.

6.3 Setup

Previous work\cite{50,102} shows that a program may demonstrate different runtime behaviour on different platforms. In order to demonstrate that both the heuristic search algorithm and the learning optimisation approach are able to improve Java performance, and that their ability is architecture-independent, the experiments were conducted within two different environments.

The first is Java(TM) 2 Runtime Environment (Standard Edition (build 1.3.0), with Java HotSpot(TM) Client VM (build 1.3.0, mixed mode)), it is installed on Linux Redhat 6.3 (kernel 2.2.19). The hardware platform used is a single Intel Celeron (533 MHz) processor with 128M RAM.

The other is Java(TM) 2 Runtime Environment (Standard Edition (build 1.4.1_01-b01), with Java HotSpot(TM) Client VM (build 1.4.1_01-b01, mixed mode)) which is installed on MS-Windows 2000. The hardware platform is Intel PentiumPro (200 MHz) processor with 96M RAM.
<table>
<thead>
<tr>
<th>Name</th>
<th>From</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel3</td>
<td>Livermor</td>
<td>inner product</td>
</tr>
<tr>
<td>kernel5</td>
<td>Livermor</td>
<td>tri-diagonal elimination, below diagonal</td>
</tr>
<tr>
<td>kernel6</td>
<td>Livermor</td>
<td>general linear recurrence equations</td>
</tr>
<tr>
<td>kernel7</td>
<td>Livermor</td>
<td>equation of state fragment</td>
</tr>
<tr>
<td>kernel8</td>
<td>Livermor</td>
<td>ADI integration</td>
</tr>
<tr>
<td>kernel9</td>
<td>Livermor</td>
<td>integrate predictors</td>
</tr>
<tr>
<td>kernel10</td>
<td>Livermor</td>
<td>difference predictors</td>
</tr>
<tr>
<td>kernel11</td>
<td>Livermor</td>
<td>first sum</td>
</tr>
<tr>
<td>kernel12</td>
<td>Livermor</td>
<td>first difference</td>
</tr>
<tr>
<td>kernel19</td>
<td>Livermor</td>
<td>general linear recurrence equations</td>
</tr>
<tr>
<td>doIteration</td>
<td>JGF</td>
<td>the double nested loop in euler::doIteration(…)</td>
</tr>
<tr>
<td>runF</td>
<td>JGF</td>
<td>euler::calculateF(…)</td>
</tr>
<tr>
<td>runG</td>
<td>JGF</td>
<td>euler::calculateG(…)</td>
</tr>
<tr>
<td>runR</td>
<td>JGF</td>
<td>euler::calculateR(…)</td>
</tr>
<tr>
<td>runS</td>
<td>JGF</td>
<td>euler::calculateStateVar(…)</td>
</tr>
<tr>
<td>mm</td>
<td></td>
<td>300x300 matrix multiplication, it is also the kernel21 of Livermore</td>
</tr>
</tbody>
</table>

Figure 6.3: Summary of Benchmarks Used

6.4 Benchmarks

The benchmarks used in the experiments were chosen from two widely-used benchmark suites, namely Java Grande Forum Benchmark Suite (JGF)[32] and Livermore[79]. Both of them contain loop- and array-intensive codes which are typical of the numerical computing in which AOF-Java is targeted at. Although Livermore does not have a Java version, the translation from its C version to Java is trivial.

Sixteen code segments were chosen from these two benchmark suites for testing. They are summarised in Figure 6.3. These benchmarks all have relatively simple program structures that can be robustly handled by the current AOF-Java prototype. Furthermore, MolDyn from JGF was used to evaluate the feasibility of using IBLO to optimise large programs with prior experience from small kernels.

6.5 Heuristic Search Results

6.5.1 Experimental approach

Experiments with the heuristic search algorithm were conducted within both Linux+Celeron and Windows+PentiumPro in the following manner. For each of the sixteen benchmarks, the heuristic search algorithm evaluated the first one hundred legal points it reached in the corresponding optimisation space. This
experiment was repeated ten times in order to minimise the impact of noise as well as to ensure that the achievements were not obtained by coincidence.

It is worth noting that the heuristic search algorithm considers more points in the search space than the legal points it evaluates. It skips those points which are found illegal. The number of these points depends on the following factors: the complexity of the benchmark program, the number of attempts in each round of the L-Search and S-Search (which is set as a constant in AOF-Java as a system configuration parameter), and the decisions randomly made during the search process. The number of points that the prototype AOF-Java needed to check before finishing its one hundred evaluations varies from four hundred to one thousand, approximately. The impact of these illegal points on the performance of AOF-Java is insignificant, because once they are found illegal, they are discarded in the current implementation. On average, the prototype AOF-Java takes less than five seconds to find a legal point during its search process for each benchmark. The vast majority of the time the search process takes is actually spent on evaluation of these points.

6.5.2 Linux+Celeron

The experimental results on Linux+Celeron are summarised in Figure 6.4. It demonstrates that the heuristic search algorithm can bring performance improvements to most of these benchmarks, except for runS whose speedup is negligible. It achieves an average speedup of 1.14 on these sixteen benchmarks.

For each of the sixteen benchmarks, the average execution time of the best version currently found by the heuristic search algorithm is plotted against the number of evaluations required to find it, the results of which are shown in Figures 6.5 - 6.12. They demonstrate that, for all the sixteen benchmarks, the heuristic search algorithm can achieve most of the speedup within a relatively small number of evaluations.

*Percentage* in Figure 6.4 is an index introduced to quantify the efficiency of the algorithm. It is defined in Equation (6.1) in order to indicate how much of the speedup it obtained within one hundred evaluations can be achieved within its first twenty attempts.

\[
\text{Percentage} = \frac{\text{After20.Speedup} - 1}{\text{After100.Speedup} - 1} \quad (6.1)
\]

Figure 6.4 shows that the heuristic search algorithm takes less than twenty evaluations to achieve more than 75% of the speedup on kernel5, kernel6, kernel7, kernel8, kernel10, kernel11, kernel12, kernel19, runF, runG, runR and mn.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>After 100</th>
<th>After 20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Speedup</td>
<td>Best</td>
</tr>
<tr>
<td>kernel3</td>
<td>5466</td>
<td>5040</td>
<td>1.09</td>
</tr>
<tr>
<td>kernel5</td>
<td>8512</td>
<td>7463</td>
<td>1.14</td>
</tr>
<tr>
<td>kernel6</td>
<td>8558</td>
<td>7318</td>
<td>1.17</td>
</tr>
<tr>
<td>kernel7</td>
<td>8542</td>
<td>8025</td>
<td>1.06</td>
</tr>
<tr>
<td>kernel8</td>
<td>5699</td>
<td>4406</td>
<td>1.29</td>
</tr>
<tr>
<td>kernel9</td>
<td>5439</td>
<td>4502</td>
<td>1.21</td>
</tr>
<tr>
<td>kernel10</td>
<td>11520</td>
<td>10200</td>
<td>1.13</td>
</tr>
<tr>
<td>kernel11</td>
<td>13991</td>
<td>9680</td>
<td>1.45</td>
</tr>
<tr>
<td>kernel12</td>
<td>10952</td>
<td>10135</td>
<td>1.08</td>
</tr>
<tr>
<td>kernel19</td>
<td>4588</td>
<td>4301</td>
<td>1.07</td>
</tr>
<tr>
<td>dOlTeration</td>
<td>12213</td>
<td>11519</td>
<td>1.06</td>
</tr>
<tr>
<td>runF</td>
<td>5504</td>
<td>5144</td>
<td>1.07</td>
</tr>
<tr>
<td>runG</td>
<td>5628</td>
<td>5149</td>
<td>1.09</td>
</tr>
<tr>
<td>runR</td>
<td>22411</td>
<td>20604</td>
<td>1.09</td>
</tr>
<tr>
<td>runS</td>
<td>15601</td>
<td>15289</td>
<td>1.02</td>
</tr>
<tr>
<td>mm</td>
<td>4625</td>
<td>3813</td>
<td>1.21</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>1.14</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.4: Heuristic search on Linux+Celeron summary

However, for kernel9 and runS, it takes over sixty evaluations, as shown in Figure 6.7 and 6.12. On average, within twenty evaluations, it can achieve 77% of the speedup it obtains within one hundred attempts. The corresponding average speedup is 1.12, compared to that of 1.14 obtained within one hundred attempts, as shown in Figure 6.4. This demonstrates the efficiency of the heuristic search algorithm.

The search results show that most of the legal points reached by the heuristic search algorithm use short and simple transformations, for instance, loop tiling only. To a certain extent, this demonstrates the effectiveness of the "simple first" strategy of the heuristic search algorithm. On the other hand, this is partly due to the fact that the comparatively simple nature of these benchmarks restrains the applicability of more complicated transformation combinations. The heuristic search algorithm biases its search towards simple transformations accordingly. This demonstrates its adaptability to the program it is to optimise.

Figures 6.5 - 6.12 show, in the error bars, the standard deviation of the ten search runs. The standard deviations are very low in most benchmarks. For example, it is negligible in kernel5, kernel6, kernel8, kernel10, kernel11, kernel19, runG, runR and mm. This indicates that in its ten runs on each of these benchmarks, the heuristic search algorithm achieves very similar results. However, the
Figure 6.5: The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel3 (upper) and kernel5 (lower) on Linux+Celeron. The standard deviations are shown in the error bars.
Figure 6.6: The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel6 (upper) and kernel7 (lower) on Linux+Celeron. The standard deviations are shown in the error bars.
Figure 6.7: The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel8 (upper) and kernel9 (lower) on Linux+Celeron. The standard deviations are shown in the error bars.
Figure 6.8: The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel10 (upper) and kernel11 (lower) on Linux+Celeron. The standard deviations are shown in the error bars.
Figure 6.9: The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel12 (upper) and kernel19 (lower) on Linux+Celeron. The standard deviations are shown in the error bars.
Figure 6.10: The solid curves show the average of the best execution time found so far during the ten heuristic searches on mm (upper) and dolteration (lower) on Linux+Celeron. The standard deviations are shown in the error bars.
Figure 6.11: The solid curves show the average of the best execution time found so far during the ten heuristic searches on runF (upper) and runG (lower) on Linux+Celeron. The standard deviations are shown in the error bars.
Figure 6.12: The solid curves show the average of the best execution time found so far during the ten heuristic searches on runR (upper) and runS (lower) on Linux+Celeron. The standard deviations are shown in the error bars.
standard deviation is less negligible on the other benchmarks, such as kernel7. The search results of kernel7 reveal that these ten runs of the heuristic search have explored the optimisation space via different paths, and reached points with different parameters. This resulted in different performance improvements. For example, one run finds, in its fourth evaluation, that tiling with tile size 38 reduces the execution time from 8595ms to 8324ms. Then in its nineteenth evaluation, it finds tiling with tile size 14 yields 8227ms. No better point is found in the remaining eighty one evaluations. However, in another run, the search finds, in its first evaluation, that tiling with tile size 5 reduces the execution time to 8442ms, then in the fifth evaluation, it finds that tiling with tile size 37 yields 8433ms. No better point is found until the eightieth evaluation where it finds that unrolling with unrolling factor 5 reduces the time further down to 8398ms.

These results indicate that, due to both the random manner of the heuristic search algorithm and the performance variety of the optimisation space, different search runs explore the space via different paths and find different good points with various performance improvements. Therefore, in some benchmarks such as kernel7 and runF, the achievements of different search runs vary, which lead to more significant standard deviations in some curves than in the others, as shown in Figure 6.5 - 6.12. This also explains why the standard deviations in the performance curves of kernel3, kernel9, dolteration and runS are negligible in early evaluations but become significant as the search goes on.

It is worth noting that although the standard deviations are more significant in some curves than in others, they are still within a low level which does not affect the trustworthiness of the above achievements.

To a certain extent, the variety amongst these results demonstrates not only the complexity of the optimisation space, but also the fact that this complexity varies from one program to another. The search results show that the optimisation space is not as smooth and flat as the curves in Figures 6.5 - 6.12 might indicate. For example, Figure 6.13 demonstrates the result of one heuristic search run on kernel6. There are many local minima in the curve, which is similar to the curves shown in Figures 3.1 and 3.2, as well as to those shown in [70]. Similar results can be found in Appendix A, which presents the result of one heuristic search run on Linux+Celeron for each of the sixteen benchmarks.

The result of one search run on kernel6 reveals that amongst the first twenty evaluations, loop tiling indicates better performance improvement potential than both loop unrolling and the combination of loop tiling and unrolling. Therefore, in the following eighty attempts, the heuristic search algorithm made its decision
Figure 6.13: The curve demonstrates one heuristic search on kernel6 on Linux + Celeron. The execution time is plotted against the number of evaluations. The large variation in kernel6’s performance caused by different transformations demonstrates the complexity of the optimisation space.
randomly but with a bias to loop tiling. It evaluated thirty-eight points that represent schedules containing loop tiling with various parameters, amongst which it finds the best schedule in its seventy-third evaluation. Among the other points it evaluates, twenty-six of them contain loop unrolling, and sixteen of them contain both loop tiling and unrolling. It visits no point that has neither tiling nor unrolling, this is because the prototype AOF-Java sets a very low weight to this bias compared to those it sets for the others. This demonstrates that although the "simple first" strategy enables the heuristic search algorithm to find good points quickly, the bias mechanism can help to find even better points, with the help of the random selection mechanism.

### 6.5.3 Windows+PentiumPro

The experimental results on Windows+PentiumPro are summarised in Figure 6.14. They demonstrate that the heuristic search algorithm can also bring performance improvement to many of these benchmarks, except for kernel19, runG and runS, where it found negligible performance improvement. It achieves an average speedup of 1.10 on these sixteen benchmarks after one hundred attempts.

Compared to the achievements on Linux+Celeron, the heuristic search algorithm achieves less performance improvement on Windows+PentiumPro, except
on kernel3, kernel9 and kernel12 where it finds higher speedups, and on kernel7, doIteration, runR and runS whose speedups are of no difference from those achieved on Linux+Celeron, although the execution time is different.

As in the previous subsection, the average execution time of the best version currently found by the heuristic search algorithm is plotted against the number of evaluations required to find it, the results of which are shown in Figure 6.15 - 6.22. They demonstrate that, for all the sixteen benchmarks, the heuristic search algorithm can achieve most of the speedup within an even smaller number of evaluations. For instance, within the first five points it evaluates, the heuristic search finds the very best points for kernel7, kernel9, kernel11 and kernel12, and the nearly-best points for kernel8, kernel19, runF, runR and mm. This is not a coincidence as the negligible standard deviations, shown by the error bars in Figure 6.15 - 6.16, indicate that all ten search runs achieve similar results.

Similarly, percentage in Figure 6.14 is defined as below.

\[
Percentage = \frac{\text{After10.Speedup} - 1}{\text{After100.Speedup} - 1}
\]  

(6.2)

Figure 6.14 shows that, on twelve out of the sixteen benchmarks, the heuristic search algorithm achieves over 80% of the speedup within ten evaluations. The average speedup obtained is 1.09, compared to that of 1.10 obtained within one hundred attempts.

The above judgment is valid not only for the benchmarks mentioned above, but for all sixteen. Their standard deviations are all negligible and within a noise level, as shown by the error bars in Figure 6.15 - 6.22. This indicates that all ten runs of the heuristic search found very similar performance improvements for them. In other words, the performance of the heuristic search algorithm is trustworthy.

The search results show that some programs’ performances are quite stable on Windows+PentiumPro, i.e. less sensitive to the transformations applied to them. For example, Figure 6.23 demonstrates the result of one heuristic search run on kernel9 on Windows+PentiumPro. It shows that, regardless of the transformations applied, its performances are almost invariant. This characteristic is not a coincidence, because the curves of the other nine runs of the heuristic search on kernel9 demonstrate the same characteristic. However, this is not always the case. Amongst all the sixteen benchmarks, only kernel9, kernel11 and kernel12 demonstrate such a characteristic, whilst the others’ curves are still highly irregular and sensitive to transformations applied. Similar results can be found in Appendix B, which presents the result of one heuristic search run on Windows+PentiumPro
Figure 6.15: The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel3 (upper) and kernel5 (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars.
Figure 6.16: The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel6 (upper) and kernel7 (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars.
Figure 6.17: The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel8 (upper) and kernel9 (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars.
Figure 6.18: The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel10 (upper) and kernel11 (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars.
Figure 6.19: The solid curves show the average of the best execution time found so far during the ten heuristic searches on kernel12 (upper) and kernel19 (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars.
Figure 6.20: The solid curves show the average of the best execution time found so far during the ten heuristic searches on mm (upper) and doIteration (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars.
Figure 6.21: The solid curves show the average of the best execution time found so far during the ten heuristic searches on runF (upper) and runG (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars.
Figure 6.22: The solid curves show the average of the best execution time found so far during the ten heuristic searches on runR (upper) and runS (lower) on Windows+PentiumPro. The standard deviations are shown in the error bars.
Figure 6.23: The curve demonstrates one heuristic search on kernel9 on Windows+PentiumPro. The execution time is plotted against the number of evaluations. The negligible variation in kernel9’s performance shows that its performance is not affected by the transformations applied.
for each of the sixteen benchmarks.

### 6.5.4 Comparison

The experimental results show that the heuristic random search algorithm presented in this thesis can bring performance improvement on both Linux+Celeron and Windows+PentiumPro environments. Its ability depends on not only the environment but also the code. On average, the algorithm achieves a speedup of 1.14 on Linux+Celeron and of 1.10 on Windows+PentiumPro. It finds higher speedup for thirteen of the sixteen benchmarks on Linux+Celeron than on Windows+PentiumPro. This indicates that, in general, iteration reordering transformations achieve better performance improvement on Linux+Celeron. The highest speedup it achieves on Linux+Celeron is 1.45 (on kernel11), compared to that of 1.37 (on kernel9) on Windows+PentiumPro. It is possible that the small L2 cache of Celeron makes the relative cost of memory latency on Linux+Celeron greater, and therefore it benefits more from cache restructuring-based transformations. The search algorithm found negligible speedup on only one benchmark (runS) on Linux+Celeron, compared to four (kernel19, runF, runG and runS) on Windows+PentiumPro.

### 6.6 Learning Results

This section evaluates the experimental results of the instance-based learning optimisation approach (IBLO) presented in the previous chapter. First, it describes how the experiments were conducted, then presents the results, before providing a comparison with the results achieved by the search algorithm.

#### 6.6.1 Experimental approach

The experiments were conducted on the sixteen benchmarks used in the previous section in a cross-validation\[94] manner, which is a standard technique also used in [87]. This means that for each benchmark, all the other fifteen benchmarks were considered as the training examples. The transformations were chosen by IBLO, and applied to the remaining benchmark for testing. Each of the sixteen benchmarks was tested in this manner on both the Linux+Celeron and Windows+PentiumPro setup.

Following IBLO, the experiments were divided into three steps. First, the sixteen benchmarks were classified into different categories by a separate program. Then, for each benchmark, five instead of one transformations were selected from
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>kernel3</td>
<td>2</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>kernel5</td>
<td>4</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>kernel6</td>
<td>3</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>kernel7</td>
<td>10</td>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>kernel8</td>
<td>33</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>kernel9</td>
<td>11</td>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>kernel10</td>
<td>20</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>kernel11</td>
<td>3</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>kernel12</td>
<td>3</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>kernel19</td>
<td>4</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>d0lteration</td>
<td>24</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>runF</td>
<td>18</td>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>runG</td>
<td>18</td>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>runR</td>
<td>116</td>
<td>6</td>
<td>T</td>
</tr>
<tr>
<td>runS</td>
<td>12</td>
<td>4</td>
<td>T</td>
</tr>
<tr>
<td>mm</td>
<td>3</td>
<td>3</td>
<td>T</td>
</tr>
</tbody>
</table>

Figure 6.24: Program features of benchmarks (before pre-processing)

the prior experience about other benchmarks most similar to it, in order to demonstrate the trustworthiness of IBLO. Finally, these transformations were tested by the prototype AOF-Java.

In order to classify the sixteen benchmarks, IBLO used the list of features presented in section 3.5 as extension to UTF. Figure 6.24 and 6.25 list all these program features of the sixteen benchmarks before and after pre-processing (as explained previously in section 5.3.2) respectively. It is worth noting that no environment feature was included. This is because the experiments were done on Linux+Celeron and Windows+PentiumPro respectively, and no cross-environment test was taken. Therefore the environmental difference was neglected.

The classification was based on the information in Figure 6.25, with equal weights given to all features except 2a (loop nest depth) and 2b (loop size) which were given a higher weight. The classification result is shown in Figure 6.26, which also demonstrates for each category, the codes it includes and its most similar category or categories.

After classifying the benchmarks, IBLO must choose suitable transformations for each of them. For each benchmark, not all the experience from all the other fifteen benchmarks were used. Only those from the most similar benchmarks were considered, the rest were simply neglected.

For each benchmark, if the category it belongs to contains other codes, IBLO
Figure 6.25: Program features of benchmarks (after pre-processing)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>kernel3</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>kernel5</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>kernel6</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>kernel7</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>kernel8</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>kernel9</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>kernel10</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>kernel11</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>kernel12</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>kernel19</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>do1eratin</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>runF</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>runG</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>runR</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>runS</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>mm</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6.26: Categories, benchmarks and the most similar categories
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Transformations from</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel3</td>
<td>kernel5</td>
</tr>
<tr>
<td>kernel5</td>
<td>kernel3</td>
</tr>
<tr>
<td>kernel6</td>
<td>kernel11, kernel12</td>
</tr>
<tr>
<td>kernel7</td>
<td>kernel9</td>
</tr>
<tr>
<td>kernel8</td>
<td>runF, runG, dolIteration</td>
</tr>
<tr>
<td>kernel9</td>
<td>kernel7, kernel10, kernel19</td>
</tr>
<tr>
<td>kernel10</td>
<td>kernel9</td>
</tr>
<tr>
<td>kernel11</td>
<td>kernel12</td>
</tr>
<tr>
<td>kernel12</td>
<td>kernel11</td>
</tr>
<tr>
<td>kernel19</td>
<td>kernel3, kernel5, kernel11</td>
</tr>
<tr>
<td>dolIteration</td>
<td>runR</td>
</tr>
<tr>
<td>runF</td>
<td>runG, runS</td>
</tr>
<tr>
<td>runG</td>
<td>runF, runS</td>
</tr>
<tr>
<td>runR</td>
<td>dolIteration</td>
</tr>
<tr>
<td>runS</td>
<td>runF, runG</td>
</tr>
<tr>
<td>mm</td>
<td>runF, runG, runS</td>
</tr>
</tbody>
</table>

Figure 6.27: Benchmarks and their transformation sources

simply chooses them as the source of its transformations. For instance, kernel3 is of category 0 which contains another benchmark kernel5, so the transformations for kernel3 are to be drawn from the experience of kernel5, and vice versa. If the benchmark is the only code in its category, IBLO chooses as its transformation source three benchmarks from its most similar categories which have the most significant performance enhancement (speedup is used here). For example, the transformations for kernel9 are drawn from kernel7, kernel10 and kernel19 in its most similar categories.

Figure 6.27 summarises the sources of transformations for all the sixteen benchmarks.

For each benchmark, the heuristic search algorithm found a list of good points during its iterative search, as shown in the previous subsection. Only the best one hundred points from all of its ten search runs were considered and the others were neglected. These one hundred points were grouped into areas according to the transformations they represent. The transformations were then randomly selected from the most significant areas, using the algorithm specified in the previous chapter. The prototype AOF-Java takes less than five seconds to complete this IBLO process for each benchmark. Each of these chosen transformations was then tested for ten times, in order to minimise the noise impact.

Due to the random manner of transformation selection, there always exists a possibility that it is not applicable for the target program, or not able to improve
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Average</th>
<th>Standard Deviation</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel3</td>
<td>1.03</td>
<td>1.02</td>
<td>1.03</td>
<td>1.04</td>
<td>1.04</td>
<td>1.03</td>
<td>0.0035</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel5</td>
<td>1.03</td>
<td>1.07</td>
<td>1.09</td>
<td>1.09</td>
<td>1.09</td>
<td>1.07</td>
<td>0.0106</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel6</td>
<td>1.15</td>
<td>1.17</td>
<td>1.14</td>
<td>1.20</td>
<td>1.13</td>
<td>1.16</td>
<td>0.0111</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel7</td>
<td>1.04</td>
<td>1.01</td>
<td>1.02</td>
<td>1.02</td>
<td>0.99</td>
<td>1.02</td>
<td>0.0075</td>
<td>4/5</td>
</tr>
<tr>
<td>kernel8</td>
<td>1.13</td>
<td>1.25</td>
<td>1.24</td>
<td>1.28</td>
<td>1.29</td>
<td>1.24</td>
<td>0.0255</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel9</td>
<td>0.99</td>
<td>1.09</td>
<td>1.08</td>
<td>1.09</td>
<td>1.08</td>
<td>1.07</td>
<td>0.0172</td>
<td>4/5</td>
</tr>
<tr>
<td>kernel10</td>
<td>0.99</td>
<td>1.09</td>
<td>1.06</td>
<td>1.09</td>
<td>1.05</td>
<td>1.06</td>
<td>0.0165</td>
<td>4/5</td>
</tr>
<tr>
<td>kernel11</td>
<td>1.06</td>
<td>1.41</td>
<td>1.40</td>
<td>1.39</td>
<td>1.39</td>
<td>1.33</td>
<td>0.0604</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel12</td>
<td>0.91</td>
<td>1.05</td>
<td>1.06</td>
<td>1.05</td>
<td>1.07</td>
<td>1.03</td>
<td>0.0258</td>
<td>4/5</td>
</tr>
<tr>
<td>kernel19</td>
<td>1.02</td>
<td>0.96</td>
<td>1.06</td>
<td>1.07</td>
<td>1.04</td>
<td>1.03</td>
<td>0.0174</td>
<td>4/5</td>
</tr>
<tr>
<td>doIteration</td>
<td>1.03</td>
<td>1.02</td>
<td>1.04</td>
<td>1.05</td>
<td>1.04</td>
<td>1.04</td>
<td>0.0053</td>
<td>5/5</td>
</tr>
<tr>
<td>runF</td>
<td>1.05</td>
<td>1.04</td>
<td>1.01</td>
<td>1.05</td>
<td>1.02</td>
<td>1.03</td>
<td>0.0072</td>
<td>5/5</td>
</tr>
<tr>
<td>runG</td>
<td>1.08</td>
<td>1.08</td>
<td>1.07</td>
<td>1.07</td>
<td>1.05</td>
<td>1.07</td>
<td>0.0049</td>
<td>5/5</td>
</tr>
<tr>
<td>runR</td>
<td>1.04</td>
<td>1.04</td>
<td>1.09</td>
<td>1.06</td>
<td>1.03</td>
<td>1.05</td>
<td>0.0096</td>
<td>5/5</td>
</tr>
<tr>
<td>runS</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.00</td>
<td>1.01</td>
<td>1.01</td>
<td>0.0020</td>
<td>5/5</td>
</tr>
<tr>
<td>mm</td>
<td>1.18</td>
<td>1.30</td>
<td>1.31</td>
<td>1.28</td>
<td>1.69</td>
<td>1.35</td>
<td>0.0784</td>
<td>5/5</td>
</tr>
</tbody>
</table>

Figure 6.28: Results of applying the chosen transformations to the benchmarks on Linux+Celeron

its performance. In order to demonstrate the trustworthiness of IBLO, five instead of one transformations were chosen and tested for each benchmark during the experiments. The results show that the achievement of IBLO is not a coincidence.

### 6.6.2 Linux+Celeron

As explained above, for each of the sixteen benchmarks, five transformations were chosen from the profile of its most similar codes’ ten heuristic search runs. Each of them was tested for ten times on Linux+Celeron in order to minimise the noise interference. The results (speedups) are shown in Figure 6.28, which shows that fifty eight out of these eighty chosen transformations can achieve a speedup higher than 1.03. This indicates that IBLO is capable of locating good transformations. In addition, for each benchmark, most of the chosen transformations make similar achievement, as the negligible Standard Deviation column indicates. However, there are still some benchmarks which IBLO fails to optimise, such as kernel3, kernel7, kernel12, kernel19, runF and runS, as the Average column in Figure 6.28 demonstrates. However, the profile information of their transformations show that many of them are in fact more effective than the Average column indicates. For instance, four out of the five transformations chosen for kernel12 improve its performance, but their achievement is offset by one transformation
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Search Best</th>
<th>Learning Best</th>
<th>Average</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel3</td>
<td>1.09</td>
<td>1.04</td>
<td>1.03</td>
<td>33%</td>
</tr>
<tr>
<td>kernel5</td>
<td>1.14</td>
<td>1.09</td>
<td>1.07</td>
<td>50%</td>
</tr>
<tr>
<td>kernel6</td>
<td>1.17</td>
<td>1.20</td>
<td>1.16</td>
<td>94%</td>
</tr>
<tr>
<td>kernel7</td>
<td>1.06</td>
<td>1.04</td>
<td>1.02</td>
<td>33%</td>
</tr>
<tr>
<td>kernel8</td>
<td>1.29</td>
<td>1.29</td>
<td>1.24</td>
<td>83%</td>
</tr>
<tr>
<td>kernel9</td>
<td>1.21</td>
<td>1.09</td>
<td>1.07</td>
<td>33%</td>
</tr>
<tr>
<td>kernel10</td>
<td>1.13</td>
<td>1.09</td>
<td>1.06</td>
<td>46%</td>
</tr>
<tr>
<td>kernel11</td>
<td>1.45</td>
<td>1.41</td>
<td>1.33</td>
<td>73%</td>
</tr>
<tr>
<td>kernel12</td>
<td>1.08</td>
<td>1.07</td>
<td>1.03</td>
<td>38%</td>
</tr>
<tr>
<td>kernel19</td>
<td>1.07</td>
<td>1.07</td>
<td>1.03</td>
<td>43%</td>
</tr>
<tr>
<td>dolteration</td>
<td>1.06</td>
<td>1.05</td>
<td>1.04</td>
<td>67%</td>
</tr>
<tr>
<td>runF</td>
<td>1.07</td>
<td>1.05</td>
<td>1.03</td>
<td>43%</td>
</tr>
<tr>
<td>runG</td>
<td>1.09</td>
<td>1.08</td>
<td>1.07</td>
<td>78%</td>
</tr>
<tr>
<td>runR</td>
<td>1.09</td>
<td>1.09</td>
<td>1.05</td>
<td>56%</td>
</tr>
<tr>
<td>runS</td>
<td>1.02</td>
<td>1.01</td>
<td>1.01</td>
<td>50%</td>
</tr>
<tr>
<td>mm</td>
<td>1.21</td>
<td>1.69</td>
<td>1.35</td>
<td>167%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>1.14</strong></td>
<td><strong>1.15</strong></td>
<td><strong>1.10</strong></td>
<td><strong>62%</strong></td>
</tr>
</tbody>
</table>

Figure 6.29: Comparison between results of the heuristic search and of learning on Linux+Celeron

(Transformation 1) which degrades its performance.

*Precision* is an index introduced in order to quantify the quality of IBLO, i.e. how well it can learn from the previous experience and select good transformations. For example, precision of 4/5 for kernel9 means that among the five transformations IBLO chooses for kernel9 (from kernel7, kernel10 and kernel19), four can improve its performance, and the other one is an improper transformation which degrades the performance.

The *Precision* column in Figure 6.28 shows that for the majority of these sixteen benchmarks, most of the transformations selected by IBLO can bring performance improvement. These results demonstrate the ability of IBLO, at least on the set of benchmarks used in the experiments.

A comparison is then made between the performance of the transformations selected by IBLO and that achieved by the heuristic search algorithm, the results of which are shown in Figure 6.29.

The *Best* and *Average* columns of *Learning* in Figure 6.29 demonstrate the best and average speedups achieved by the five transformations chosen for each of these sixteen benchmarks. Compared to the achievement of the heuristic search algorithm (as shown in the *Best* column of *Search*), these achievements are less
significant for most benchmarks, except for mm and the best transformation chosen for kernel6.

The experimental results of the transformations which IBLO selected for mm reveal that the extra performance improvement comes from a transformation of the combination of loop tiling and unrolling. This direction is given a comparatively low bias in the heuristic search. Therefore, in the ten runs, there are just a relatively small number of points which try this transformation combination with various parameter combinations and other additional transformations. The results of the heuristic search on mm show that some of these points are of proper parameter combinations which can bring more significant performance improvements than the others. However, when the achievements are averaged over the ten search runs, the significant achievement obtained in a run is offset by the results of the other runs which failed to locate good points of proper parameter combinations.

*Percentage* is another index introduced to quantify the ability of IBLO. It is defined as below in order to indicate how much IBLO can achieve, with the transformation it chose from its prior experience, the performance improvement found by the heuristic search algorithm.

\[
\text{Percentage} = \frac{\text{Learning.Average} - 1}{\text{Search.Best} - 1}
\]

(6.3)

The *Percentage* column in Figure 6.29 show that for nine of these sixteen benchmarks, IBLO can, within just one attempt, explore over 50% of the performance improvement potential which costs the heuristic search algorithm around twenty attempts to achieve and more to confirm (as discussed in the previous section). In the cases of kernel12, kernel19 and runF, whose *Percentage* are below 50%, many transformations IBLO selected achieve a higher percentage of the performance improvement, as Figure 6.28 indicates. This demonstrates the ability of IBLO on Linux+Celeron.

### 6.6.3 Windows+PentiumPro

The same experiments were repeated in the Windows+PentiumPro environment, and the results are presented in Figure 6.30.

Similar to the results on Linux+Celeron, the majority of these chosen transformations can improve the performance for the corresponding benchmarks. However, even the best transformation fails to improve that of kernel19, runG and runS, and the average performance improvements on kernel6 and runF are negligible. It is worth noting that the heuristic search algorithm finds negligible
performance improvements for all of them except kernel6, as shown in Figure 6.14. This indicates that there may be little performance improvement space for these codes on Windows+PentiumPro. Therefore, it is understandable that the probability of finding a good transformation in the huge optimisation space is relatively low. Furthermore, only one transformation improves kernel6, and the improvement is negligible. The others degrade its performance, because the random transformation selection failed to select transformations of proper parameter combinations.

For the other eleven benchmarks, all five transformations chosen by IBLO can improve their performance, as the Precision column in Figure 6.30 demonstrates. The improvements vary from negligible (for example, 1.03 for dolteration) to significant (for example, 1.35 for kernel9).

The extra performance improvement for mm on Linux+Celeron does not occur on Windows+PentiumPro. This may due to a number of factors. For instance, the combination of loop tiling and unrolling might not be able to improve the performance any further in this Windows+PentiumPro setup, or IBLO fails to locate a good point with proper parameter combination via its random transformation selection, because the distribution of such good points is unpredictable in such a highly irregular optimisation space. In addition, there is no concrete evidence about the existence of points with higher speedup in this space. However, IBLO has already found good transformations which lead to good performance improvement similar to what the heuristic search algorithm can find.

Comparison is also made between the achievements made by the heuristic search algorithm and those by IBLO, the results of which are shown in Figure 6.31. Among these sixteen benchmarks, the heuristic search algorithm can find performance improvement on twelve of them on Windows+PentiumPro. IBLO achieves similar results for all of them except kernel6, as discussed above. It is worth noting that, in eight of the other eleven benchmarks, more than 85% of what the heuristic search algorithm has achieved can be obtained by IBLO within just one attempt, and over 60% for the other three (dolteration, runR and mm), as the Percentage column in Figure 6.31 indicates. In brief, the difference between their performances is negligible. This indicates that if properly trained, IBLO can be as competent as the heuristic random search algorithm.

### 6.6.4 Comparison

IBLO achieves an average speedup of 1.10 on Linux+Celeron and 1.09 on Windows+PentiumPro. The differences between the speedups obtained on Linux+Celeron
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Average</th>
<th>Standard Deviation</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel3</td>
<td>1.17</td>
<td>1.17</td>
<td>1.17</td>
<td>1.17</td>
<td>1.17</td>
<td>1.17</td>
<td>0.0000</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel5</td>
<td>1.10</td>
<td>1.10</td>
<td>1.10</td>
<td>1.10</td>
<td>1.10</td>
<td>1.10</td>
<td>0.0000</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel6</td>
<td>0.95</td>
<td>0.95</td>
<td>0.98</td>
<td>0.98</td>
<td>1.03</td>
<td>0.98</td>
<td>0.0041</td>
<td>1/5</td>
</tr>
<tr>
<td>kernel7</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
<td>0.0000</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel8</td>
<td>1.12</td>
<td>1.10</td>
<td>1.12</td>
<td>1.13</td>
<td>1.13</td>
<td>1.12</td>
<td>0.0050</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel9</td>
<td>1.35</td>
<td>1.35</td>
<td>1.35</td>
<td>1.34</td>
<td>1.35</td>
<td>1.35</td>
<td>0.0020</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel10</td>
<td>1.07</td>
<td>1.05</td>
<td>1.06</td>
<td>1.06</td>
<td>1.06</td>
<td>1.06</td>
<td>0.0028</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel11</td>
<td>1.18</td>
<td>1.18</td>
<td>1.18</td>
<td>1.18</td>
<td>1.18</td>
<td>1.18</td>
<td>0.0000</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel12</td>
<td>1.18</td>
<td>1.18</td>
<td>1.18</td>
<td>1.18</td>
<td>1.18</td>
<td>1.18</td>
<td>0.0000</td>
<td>5/5</td>
</tr>
<tr>
<td>kernel19</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
<td>0.0040</td>
<td>0/5</td>
</tr>
<tr>
<td>dolteration</td>
<td>1.03</td>
<td>1.02</td>
<td>1.02</td>
<td>1.03</td>
<td>1.03</td>
<td>1.03</td>
<td>0.0028</td>
<td>5/5</td>
</tr>
<tr>
<td>runF</td>
<td>1.01</td>
<td>0.99</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.0035</td>
<td>2/5</td>
</tr>
<tr>
<td>runG</td>
<td>0.99</td>
<td>0.99</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>0.99</td>
<td>0.0028</td>
<td>0/5</td>
</tr>
<tr>
<td>runR</td>
<td>1.04</td>
<td>1.05</td>
<td>1.09</td>
<td>1.08</td>
<td>1.08</td>
<td>1.07</td>
<td>0.0087</td>
<td>5/5</td>
</tr>
<tr>
<td>runS</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.0000</td>
<td>0/5</td>
</tr>
<tr>
<td>mm</td>
<td>1.10</td>
<td>1.11</td>
<td>1.10</td>
<td>1.10</td>
<td>1.10</td>
<td>1.10</td>
<td>0.0020</td>
<td>5/5</td>
</tr>
</tbody>
</table>

Figure 6.30: Results of applying the chosen transformations to the benchmarks on Windows+PentiumPro

and on Windows+PentiumPro are significant for seven benchmarks. For example it reaches as high as 1.35-1.07=0.28 for kernel9. IBLO achieves significantly higher speedup for kernel3, kernel9 and kernel12 on Windows+PentiumPro than on Linux+Celeron. For the remaining four (kernel6, kernel8, kernel11 and mm), the average speedup is higher on Linux+Celeron than on Windows+PentiumPro.

A comparison between the Precision columns in Figure 6.28 and 6.30 indicates that IBLO has chosen better transformations from its prior experience on Linux+Celeron than from that on Windows+PentiumPro. More transformations chosen in the Linux+Celeron cases are good ones, whilst it has a low precision on Windows+PentiumPro case for five benchmarks (kernel6, kernel19, runF, runG and runS, as discussed in 6.6.3).

### 6.7 Large Benchmark Results

*MolDyn* from *Java Grande Benchmark Suite* (JGF)[32] is chosen as the large benchmark to evaluate the feasibility of using prior experience obtained from small kernels to optimise large programs using the learning optimisation approach presented in this thesis.

The loops in *MolDyn* were classified into various categories by the IBLO clas-
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Search Best</th>
<th>Learning Best</th>
<th>Learning Average</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel3</td>
<td>1.18</td>
<td>1.17</td>
<td>1.17</td>
<td>94%</td>
</tr>
<tr>
<td>kernel5</td>
<td>1.10</td>
<td>1.10</td>
<td>1.10</td>
<td>100%</td>
</tr>
<tr>
<td>kernel6</td>
<td>1.09</td>
<td>1.03</td>
<td>0.98</td>
<td>-</td>
</tr>
<tr>
<td>kernel7</td>
<td>1.05</td>
<td>1.05</td>
<td>1.05</td>
<td>100%</td>
</tr>
<tr>
<td>kernel8</td>
<td>1.14</td>
<td>1.13</td>
<td>1.12</td>
<td>86%</td>
</tr>
<tr>
<td>kernel9</td>
<td>1.37</td>
<td>1.35</td>
<td>1.35</td>
<td>95%</td>
</tr>
<tr>
<td>kernel10</td>
<td>1.06</td>
<td>1.07</td>
<td>1.06</td>
<td>100%</td>
</tr>
<tr>
<td>kernel11</td>
<td>1.18</td>
<td>1.18</td>
<td>1.18</td>
<td>100%</td>
</tr>
<tr>
<td>kernel12</td>
<td>1.19</td>
<td>1.18</td>
<td>1.18</td>
<td>95%</td>
</tr>
<tr>
<td>kernel19</td>
<td>1.01</td>
<td>1.00</td>
<td>0.99</td>
<td>-</td>
</tr>
<tr>
<td>dolteration</td>
<td>1.05</td>
<td>1.03</td>
<td>1.03</td>
<td>60%</td>
</tr>
<tr>
<td>runF</td>
<td>1.02</td>
<td>1.01</td>
<td>1.00</td>
<td>0%</td>
</tr>
<tr>
<td>runG</td>
<td>1.01</td>
<td>1.00</td>
<td>0.99</td>
<td>-</td>
</tr>
<tr>
<td>runR</td>
<td>1.09</td>
<td>1.09</td>
<td>1.07</td>
<td>78%</td>
</tr>
<tr>
<td>runS</td>
<td>1.01</td>
<td>0.99</td>
<td>0.99</td>
<td>-</td>
</tr>
<tr>
<td>mm</td>
<td>1.14</td>
<td>1.11</td>
<td>1.10</td>
<td>71%</td>
</tr>
<tr>
<td>Average</td>
<td>1.10</td>
<td>1.09</td>
<td>1.09</td>
<td>62%</td>
</tr>
</tbody>
</table>

Figure 6.31: Comparison between results of the heuristic search and of learning on Windows+PentiumPro

sification algorithm, as explained in the previous section. For each of these categories, its most similar category was chosen from those collected from the previous sixteen benchmarks (as demonstrated in Figure 6.26). Proper transformations were then chosen from those collected for these categories in Linux and Windows respectively, before they were applied to the corresponding loops in MolDyn.

In order to minimise the noise interference, both the original MolDyn and its optimised version were tested ten times in both Linux and Windows environments. Experiments show that loop tiling reduces the execution time from 53957ms to 50233ms on Linux, i.e. with a speedup of 1.07, and from 79781ms to 79712ms on Windows, i.e. with a negligible speedup.

These results indicate that IBLO is capable of using the prior experience obtained from small kernels to improve the performance of large programs.

6.8 Summary

This chapter has presented the experimental results of both the heuristic random search algorithm and the instance-based learning optimisation approach within two different environments. First, a brief introduction was given to the prototype
AOF-Java, the experimental setup and the benchmarks used. Experimental results of the heuristic search algorithm have been presented and discussed before those of IBLO. It has given the experimental results of applying IBLO to a large benchmark at the end of the chapter.

The experimental results have shown that the heuristic search algorithm can find, within a relatively small number of attempts, good points in the large optimisation space specified in chapter 3. In addition, IBLO is capable of finding good transformations for a given program from its experience with other programs similar to it. These results have demonstrated the prospect of applying proper machine learning techniques to an optimising compiler.

Previous work[46][50][51][70][71] has used various techniques to improve the performance of some of these sixteen benchmarks within different environments. However, they are not directly comparable as they target different languages (C and Fortran). There is no comparable optimising Java compiler work on uni-processor. The nearest comparable project, NINJA[88][89][90][91][93], examines a different benchmark suite, unfortunately the results can not be compared.
Chapter 7
Conclusion

This thesis presents an architecture- and language-independent program optimisation approach which uses iterative optimisation, heuristic search and machine learning techniques. This approach can adapt itself to various architectures and achieve portable performance improvement.

A summary of the thesis and its contribution is given in section 7.1, followed by a critical review of the overall approach in section 7.2. A brief description of future work is given in the last section.

7.1 Summary

The introduction chapter described the demand for portable performance improvement, which is generated by the new computational requirement and the increasing complexity and rapid change of modern computer architectures. A learning optimisation approach was proposed to meet this challenge. This approach was summarised and its potentials were briefly discussed in this chapter.

Chapter 2 gave a review of related work. It includes Java optimisation, adaptive optimisation, iterative optimisation, systematic exploration of optimisation spaces, and some previous projects which apply machine learning techniques to specific optimisation problems.

The optimisation work can be considered as searching in an optimisation space for the optimal point(s). Chapter 3 specified the problem that this thesis aims to address, and demonstrated its complexity by way of an example. It gave a review of the Unified Transformation Framework (UTF) and specified a UTF-based optimisation space with the help of an exhaustive scan algorithm.

Chapter 4 presented a heuristic random search algorithm which is capable of exploring the above optimisation space effectively. This algorithm is the foundation and first step of the learning optimisation approach, in that it provides the
latter with training examples to learn from. It explores the optimisation space by carrying out L-Search and S-Search in roughly alternating manner. These two searches aim to explore the space in different directions. This chapter first described the search strategies of the heuristic random search algorithm, then specified this algorithm in depth, before comparing it with other algorithms described in previous work.

Chapter 5 presented the learning optimisation approach in depth. First it explained the concept of a learning compiler and provided a review of some applicable machine learning approaches. Then it explained in depth the learning optimisation approach which is based on instance-based learning. It concluded with a brief comparison of this approach and related work reviewed in chapter 2.

Chapter 6 gave a brief review of the prototype AOF-Java framework. It presented and analysed the experimental results of both the heuristic search algorithm and the learning optimisation approach. Experimental results showed that the heuristic search algorithm can quickly locate good points in the large optimisation space specified before, and the instance-based learning optimisation approach can select good transformations for a program based on similar programs.

The overall contributions of this thesis are the UTF-based heuristic search algorithm and the instance-based learning optimisation approach, both of which are portable and independent of architecture, environment and language. Instead of replacing the existing optimisation techniques, learning optimisation aims to provide an architecture and environment-independent steering module in order to further explore the potential of these existing techniques, whilst leaving the architecture-specific optimisation to the underlying compiler. In addition, it has the ability to acquire knowledge from previous attempts and apply the knowledge when necessary. This enables the compiler to adapt to the environment in order to achieve portable high performance in modern computing environments quickly. Experimental results have demonstrated the ability of the search algorithm and the prospect of optimisation via a machine learning approach. It shows that such a learning optimisation approach is feasible and capable of bringing portable performance improvement quickly.

7.2 Critique

One of the criticisms is the selection of program and environment features. The current AOF-Java prototype does not consider any environment feature, it uses
a small set of program features which are similar to those used in [86]. Both the program and its runtime environment can be better specified if more features are included. As the current approach has demonstrated the effectiveness of instance-based learning, AOF-Java can be further enhanced by the introduction of more program and environment features which describe different aspects of the runtime context and the target program.

However, a larger feature set will result in significantly more combinations and, therefore, become expensive to process. The current Cartesian-product-like approach to classification and storage is inefficient since it does not scale well. In addition, the linear search of category retrieval could be replaced by a more efficient approach such as hash table search.

With a larger set of program and environment features, one problem to consider is that these features are not equally important. Their contributions to the transformation selection are likely to be different. Furthermore, different transformations usually focus on different subsets of the available features. It is believed that this information can only be accumulated gradually in a long run during which the compiler encounters many programs of various categories. The current instance-based learning optimisation approach (IBLO) does not provide a flexible solution to this problem. An algorithm is needed to distinguish more important features from the others and assign different weights to them in order to improve the efficiency in classification and retrieval. It should be adaptive in order to correctly classify various programs it is given. For instance, it might happen that all of the training examples the compiler receives so far share a common value of one specific feature, the classification algorithm may therefore consider this feature is of little significance and give it a low weight. When a new program arrives later with a different value of this feature, the compiler might classify it incorrectly and choose an inappropriate transformation. Adaptability is needed to address this problem.

Another criticism is that the heuristic random search algorithm does not use any available program analyses. It treats the target program as a black box and relies only on the bias mechanism to direct the random search. Therefore, the optimisation space remains intact throughout the search, although could be shrunk if program information becomes available. Program analyses can provide the compiler with useful in-depth information about the program. For instance, if data dependency analysis finds out that a specific statement reordering is illegal, the compiler may learn and try to avoid it in its future efforts. Although the experimental results show that the heuristic search algorithm can find good points
quickly, it checks more illegal points than those it evaluates, as explained in chapter 6. With such program information, these illegal points could have been removed from the space during the search progress, and the effort of checking them could be saved. Therefore, it is believed that the efficiency of both the heuristic search algorithm and IBLO could be further improved if program analyses are integrated into the system. This benefit could become more significant when more transformations are integrated into the compiler.

A potential problem of IBLO is that, no matter how long the learning process may last, the space of potential transformations remains unchanged. This is because the approach can only select transformation from existing ones from this space, and no new point will be generated to enlarge the transformation base, although the approach may create more and more new categories for new programs it meet. This problem will become serious if only a small number of training examples are used to train the compiler which will later meet various programs.

A solution to this problem is a hybrid approach which integrates both the heuristic search algorithm and IBLO. When a new program arises, the framework uses IBLO to find a proper transformation. If the budget allowed, the compiler can use it as the starting point of a heuristic search, in the similar way to the in-depth L-Search. The difference is that good loop vectors are chosen by IBLO rather than directly from the profile database. This hybrid approach can therefore keep enlarging its knowledge base with new transformations it find during the process. It has not been implemented in this thesis.

7.3 Future Work

Obviously the first step is to introduce more program and environment features into the AOF-Java. Solutions must be found to the problems pointed out in the previous section. One problem to solve is how to collect these feature information automatically. Linear dimension reduction and principle component analysis[82] are good starting points to consider feature distinction.

There exist many useful transformations other than iteration reordering transformations considered in this thesis. Their abilities to improve performance have been demonstrated by previous work[4][14][17][21][35][36][38][88][112][121][129]. For example, common subexpression elimination and method inlining are two useful transformations that have been shown to be effective and widely used in optimising compilers. It is worthwhile considering how to enhance the current
learning optimisation approach with these transformations. Considering the diversity of these transformations, the key issue of integration is to find a unified but still flexible way to represent them.

In addition, efforts should be put on integrating program analyses such as data dependency analysis into the compiler. This could further improve the performance of both the heuristic search algorithm and the learning optimisation approach, as explained in the previous section. The key issue is how to represent the results obtained from these analyses, so that they could be understood and used by the learning optimisation approach.

It is important that a compiler keeps obtaining new knowledge of optimisation through all of its previous efforts rather than just from the training examples. The hybrid approach proposed in the previous section is vital to achieve this. It should be implemented in order to coordinate both the heuristic random search algorithm and the learning optimisation approach in search for better performance.

Another interesting area is to test the ability of this learning optimisation approach under various performance metrics. Currently, execution time is used. However, under different circumstances, different performance metrics may be needed for different optimisation targets. For instance code size and power consumption in the case of optimisation for embedded systems. Profile-based optimisation has been shown[81] to be inadequate for real-world applications on metrics such as memory usage and disk performance, even with extensive training cases. Theoretically, this learning optimisation approach can work with any given metric or metrics. It is worthwhile exploring how the learning optimisation approach responds to different performance metrics. Furthermore, if there is more than one metric to consider, a way must be found to model the overall target function in a proper manner. Flexibility and extensibility are two key concerns here.

Furthermore, the current learning optimisation approach only learns from positive training examples. It is obvious that negative examples also provide useful information. For example, due to the legality constraints, there may be many more illegal mappings than legal mappings. The learning compiler could be expected to obtain useful information also from these illegal mappings in order not to generate more later. Currently, it uses a biased random selection algorithm to achieve this. This naive approach works but responds slowly. A more efficient learning mechanism should be developed.

Finally, the selection of training examples has a significant impact in the classification of the IBLO. The knowledge will be biased if the training examples are obtained from just a small set of similar programs, i.e. from a small region
of the whole optimisation space. Biased knowledge may lead to improper transformations when the compiler meets a different program. An approach must be developed to choose proper training examples so that IBLO can learn properly and collect unbiased and comprehensive experience.
Appendix A

Heuristic Search in Linux

The diagrams listed below demonstrate one heuristic search on each of the sixteen benchmarks in Linux. The execution time is plotted against the number of evaluations in each of these diagrams. The variations in performance caused by different transformations demonstrate the complexity of the corresponding optimisation spaces.
"kernel5.linux.data"
The graphs show the execution time (ms) for two different kernel data files: "kernel19.linux.data" and "mm.linux.data". Each graph plots the execution time against the number of evaluations. The data for "kernel19.linux.data" is represented by a solid line, while "mm.linux.data" is represented by a dashed line. The graphs indicate variability in execution time across different evaluations.
execution time (ms) | no. of evaluations
--- | ---
*runG.linux.data*

execution time (ms) | no. of evaluations
--- | ---
*runR.linux.data*
Appendix B

Heuristic Search in Windows

The diagrams listed below demonstrate one heuristic search on each of the sixteen benchmarks in Windows. The execution time is plotted against the number of evaluations in each of these diagrams. The variations in performance caused by different transformations demonstrate the complexity of the corresponding optimisation spaces.
The graphs show the execution time (ms) against the number of evaluations for two different datasets: "kernel9.win.data" and "kernel10.win.data". The x-axis represents the number of evaluations, and the y-axis represents the execution time in milliseconds. The data points are scattered across the graphs, indicating variability in execution times for different numbers of evaluations.
170
Bibliography


174


[31] Z. Budimlic and K. Kennedy. Almost-whole-program compilation, proc. of the 3rd Workshop on Java for High Performance Computing (in conjunction
with the 15th ACM International Conference on Supercomputing (ICS’01)), 2001.


[103] M. Poleto, D. Engler and M. Kaashock. tcc: a system for fast, flexible, and high-level dynamic code generation, proc. of Workshop on Compiler Support for System Software (WCSSS96), 1996;


183


185