Iterative Compilation and Performance Prediction for Numerical Applications

Grigori G. Fursin

Doctor of Philosophy

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2004

Abstract

As the current rate of improvement in processor performance far exceeds the rate of memory performance, memory latency is the dominant overhead in many performance critical applications. In many cases, automatic compiler-based approaches to improving memory performance are limited and programmers frequently resort to manual optimisation techniques. However, this process is tedious and time-consuming. Furthermore, a diverse range of a rapidly evolving hardware makes the optimisation process even more complex. It is often hard to predict the potential benefits from different optimisations and there are no simple criteria to stop optimisations i.e. when optimal memory performance has been achieved or sufficiently approached.

This thesis presents a platform independent optimisation approach for numerical applications based on iterative feedback-directed program restructuring using a new reasonably fast and accurate performance prediction technique for guiding optimisations. New strategies for searching the optimisation space, by means of profiling to find the best possible program variant, have been developed. These strategies have been evaluated using a range of kernels and programs on different platforms and operating systems. A significant performance improvement has been achieved using new approaches when compared to the state-of-the-art native static and platform-specific feedback directed compilers.

Acknowledgements

I would like to thank my supervisor Dr. Michael O'Boyle for the overwhelming support during all the hard years of this research and for providing a great working environment. I would also like to thank Prof. Nigel Topham for his important guidance at the beginning of the project, Prof. Olivier Temam, Dr. Peter Knijnenburg and members of the MHAOTEU project for the collaboration and fruitful discussions.

I am grateful to Monika, Dyane and Margaret for their great administrative support.

I would like to thank Shun, Bjoern, Tom and Tim for the interesting discussions during lunch breaks.

I am grateful to all my friends who helped me to switch off my work occasionally and relax, particularly to Viki, Laura, Ernest, Joe, Annemieke, Pille, Paulo, Aghlab, Gaurav, Melissa, Hachemy, Katarina, Ghassan, Takeshi, Emilio, Atif, Yannos, Lucia and all my partners in sport activities.

Finally, special thanks to Eglantine and Georgios for keeping me out of the misery during difficult times, and to my parents and brother Leonid for the moral support.

Declaration

I declare that all the research and developed software presented in this thesis is my own, unless stated otherwise in the text. Some of the material used in this thesis has been published in the following papers:

[FOT+04] G. Fursin, M. O'Boyle, O. Temam, and G. Watts. Fast and accurate method for determining a lower bound on execution time. *Concurrency Practice and Experience*, 16(2-3), pages 271-292, 2004.

[FBK03] G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg. Evaluating Iterative Compilation. *Accepted for publication in "Springer Lecture Notes in Computer Science"*.

[FBK02] G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg. Evaluating Iterative Compilation. *Proceedings of Languages and Compilers for Parallel Computing (LCPC'02),* pages 305-315, 2002.

[FOT+01] G. Fursin, M. O'Boyle, O. Temam, and G. Watts. Fast and accurate evaluation of memory performance upper-bound. *Proceedings of Compilers for Parallel Computers (CPC'01),* pages 163-171, June 2001.

January, 2004

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter briefly describes the research area, the contributions and the structure of this thesis.

1.1 The problem

Considerable progress has been made in processor technology in the last 30 years. Early processors were simple 4/8-bit in-order execution chips with working frequencies of several megahertz, supporting only direct-addressed memory of several hundred kilobytes. Currently, however, they are complex 32/64-bit devices working at gigahertz frequencies with the support of out-of-order parallel multiple instruction execution, value prediction, speculation and virtual memory support. The sole motivation behind these advances is to make the processor perform computations faster. Naturally, the amount of data to process has also grown. This data is kept in main memory and is accessed by the processor as and when needed.

One of the major problems in current computing systems is that the memory cannot supply data to the processor immediately on request due to its physical size and the speed of signal propagation. This leads to a mismatch between processor and memory performance. It was observed that while microprocessor performance has improved by approximately 55% per year since 1987, memory performance has only improved by 7% per year [HP96]. This leads to the processor-memory bottleneck; no matter how fast the processor is, the overall performance of the computing system is limited by the speed of memory.

The most common solution to this problem is based on the introduction of intermediate smaller, but faster layers of memory, known as cache memory, between the processor and main memory [Smi82]. Caches are designed to exploit program locality [MB76] and are based on the two following observations: a) a memory location recently referenced is likely to be referenced again soon and b) a memory location adjacent to a referenced location is likely to be referenced soon. In practice,

however, programs may not exhibit this property. In this case, the task of restructuring of the data layout in memory or transforming the program to exploit locality has to be performed either manually by the programmer or automatically by the compiler.

Modifying the program manually is tedious, time consuming and requires a good knowledge of the underlying hardware. Furthermore, if the program needs to be ported to a new platform, it has to be optimised once again to reflect the new hardware parameters, which may require many man-hours and hence is economically expensive. Conversely, compilers attempt to solve this problem by utilising static models of different platforms and transforming the code to match the particular hardware platform. Nevertheless, due to the complexity of the memory and processor architecture and for reasons of tractable analysis, compilers have to assume a significantly simplified machine model. In addition to this, the lack of important run-time information such as loop bounds and branches taken, means that compiler memory optimisations often fail to achieve performance improvements.

One of the techniques introduced to reflect the importance of run-time information is profile-directed compilation [PH90]. It is a dynamic optimisation process, which is performed in two steps. In the first step, the optimised program is instrumented and executed to collect certain run-time parameters. In the second step, the program is optimised according to the information obtained. Yet, most current profile-directed optimisations attempt to improve instruction cache use or enable better branch prediction whilst ignoring data cache usage, which may not improve the overall program performance if a memory bottleneck is present.

A further weakness of current techniques is the inability to determine the potential benefit from an optimisation. Performance prediction techniques are usually based on a simplified hardware model and are inaccurate, or based on simulators that are extremely slow, sometimes by several orders of magnitude in comparison with the original program execution time. Alternatively, hardware counters can be used, but they often mispredict performance on current superscalar out-of-order execution processors. For example, a program may generate many cache misses that will be detected by hardware counters. This will lead to an assumption that memory optimisations are beneficial for this program, but memory access delays may be

hidden by other calculations performed in parallel and in this case memory optimisations will fail in gaining performance. However, knowing the potential performance improvement before optimising the code is important for judging the amount of effort worth expending.

1.2 Contributions

Three major contributions to the above problems are presented in this thesis. An iterative feedback assisted optimisation approach is presented. It is based on searching for the best possible program transformations in a large optimisation space. Unlike other search optimisation techniques that use some heuristics to analyse and optimise small kernels, it can successfully optimise large applications by applying transformations in a smart phase order to cut down the search space. This approach, while being slow and requiring thousands of runs of the transformed program, achieves a considerable performance improvement over state-of-the-art compilers. Considering that the set of transformations used in this approach is specially chosen to be the same or smaller than in used compilers, it demonstrates that current optimisers fail to find the best possible transformations statically.

A new performance prediction technique for estimating the lower bound on program execution time is then presented. This is a dynamic, reasonably fast and accurate approach, based on transforming all array references into scalar references to remove cache misses, and profiling the new code.

Finally, an approach for reducing the iterative compilation time dramatically is presented. It uses the performance prediction technique to detect sections of the program that may potentially benefit from optimisations and applies a random iterative transformation search to those sections. This can reduce the number of iterations by two orders of magnitude in comparison with the basic iterative search, thus making iterative compilation a superior and realistic option over the current static or profile-directed optimisations.

The developed techniques have been implemented inside a cross-platform toolset, evaluated on two distinct RISC and CISC platforms using a variety of kernels and benchmarks and compared with the current native state-of-the-art compilers.

1.3 Thesis structure

This thesis has the following structure. Chapter 2 surveys various processor designs and advances in the semiconductor technology. It describes and analyses techniques for exploiting instruction level parallelism such as pipelining and multiple instruction issue with out-of-order execution. This chapter further presents the evolution of the memory hierarchy and describes various cache designs that exploit locality to reduce memory access time. It also contains an introduction to basic compilation and optimisation techniques.

Chapter 3 presents related work on memory optimisations. It starts with introducing formal notations for describing loops and data accesses, and then presents models to unify and ease program transformations and data locality analysis. This is followed by a description of various static methods for analysing and improving cache utilisation for a broad range of programs. This chapter concludes by presenting multiple dynamic techniques for profiling and optimising program performance.

Chapter 4 describes a new platform-independent iterative optimisation approach that is able to outperform current state-of-the-art commercial compilers with both static and feedback-directed optimisations enabled. This chapter analyses the influence of array padding, loop tiling and loop unrolling transformations on the program performance in detail, and examines the reasons why static optimisation approaches often fail in improving performance or can even degrade it. Experimental results show considerable performance improvements after using this iterative approach for two kernels and eight SPEC benchmarks on two platforms. However, the major drawback of iterative compilation is its excessive optimisation time. Thousands of executions of program variants are often needed, which may not be tolerable for general-purpose computing. Therefore, the two following chapters present techniques to reduce iterative compilation time.

Chapter 5 presents a new performance prediction technique that can provide information about whether program segments have the potential for performance improvement or not. This platform-independent technique transforms the original program at the assembler level in such a way that the new program behaves as if

there were no cache misses occurring. Profiling the original and transformed programs and comparing the difference in the execution time shows the potential for performance improvement. This technique is reasonably fast and accurate as no simulations are involved and no approximations are used. It is compared to other existing methods and it is shown that many of these methods, which are based on counting the number of cache misses, give inaccurate predictions on modern superscalar processors with out-of-order execution. Performance prediction can reduce the iterative compilation search space by removing loop nests that do not have any potential for performance improvement from the search.

Chapter 6 presents a new iterative compilation approach that combines performance prediction and random search, thus considerably reducing the search space. Using this optimisation technique reduces the number of program executions to less than a hundred while still obtaining considerable performance improvement. This makes iterative compilation a realistic approach for general-purpose optimisation. A comparison with other techniques is presented at the end of this chapter. Finally, chapter 7 summarises the results achieved and outlines future work.

Chapter 2

Background

This chapter briefly surveys trends in processor design; describes memory design evolution and summarises basic compiler technology. It starts with a short description of the first microprocessor architecture followed by a review of major hardware design changes to improve its performance. These changes are possibly due to the advances in semiconductor technology and the gradually increasing number of transistors on the chip. Pipelined superscalar processors with out-of-order execution to exploit instruction level parallelism are discussed. The evolution of memory design to improve the speed of data access is further presented and various cache designs to exploit program locality are described. Finally, basic program compilation and optimisation methods are discussed.

2.1 Processor architecture

It is important to know the architecture of the platform in order to effectively optimise programs so that all platform specific features are used in the best possible way. Therefore, this section describes the architecture of the modern processors used in this thesis and presents the major techniques used to improve their performance. It briefly surveys processor evolution and describes pipelining and parallel out-of-order execution of instructions.

2.1.1 Processor design evolution

The history of microprocessors dates back to 1971 when the world's first microprocessor, the Intel 4004, was introduced [FHM+96]. The major difference between this processor and other computing devices was that all its components were assembled on a single semiconductor chip. The design of this processor is shown in figure 2.1. It is based on the von Neumann architecture [BGN63] that uses the same storage for both data and instructions, fetching and executing instructions one by

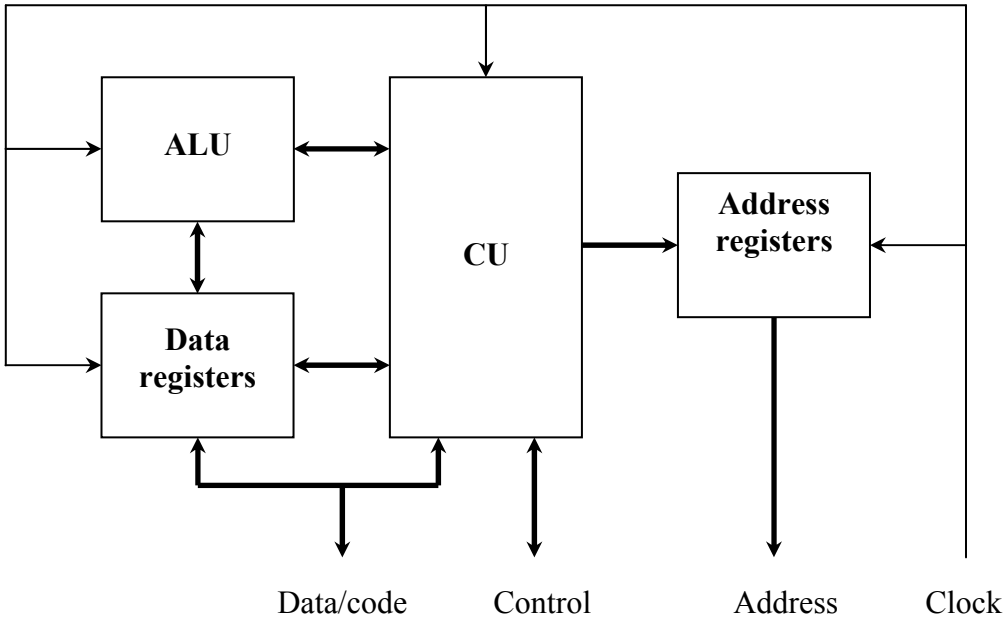


Figure 2.1: Von Neumann processor architecture

one. It consists of registers that contain temporal data and memory addresses, a functional unit or ALU (arithmetic and logic unit) that performs mathematical operations and a CU (control unit) that fetches an instruction, decodes it according to the instruction set and controls its execution. Typically, instruction execution in a von Neumann processor occurs in five stages: fetching the instruction, decoding the instruction, loading data from memory or register, performing an operation and storing the result in a register or memory. Thin arrows in figure 2.1 show the propagation of the synchronisation clock signal. Thick arrows in this figure represent wide bus connections consisting of more then one signal that interconnect all processor components. The processor also communicates with external devices such as memory using external data/code, control and address buses.

The Intel 4004 has a simple design by today's measures. It processes data in 4 bits, has sixteen 4-bit general-purpose registers, can address up to 4 KB of data memory and has a clock speed of 108 KHz. Nevertheless, most of its components can still be found in current mainstream SISD (single instruction single data) computing systems, according to Flynn's computer architecture classification [Fly72]. Other types of computing systems such as SIMD (single instruction multiple data) and MIMD (multiple instruction multiple data) are used in parallel machines

and surveyed in [Dun90], however, they are beyond the scope of this thesis. SISD computers are further classified by their instruction set. If they support a large number of complex instructions covering as many operations as possible, they are called CISC (complex instruction set computer) computers. If they support a minimal instruction set covering only the most commonly used instructions, they are called RISC (reduced instruction set computer) computers. Differences between these architectures are described further in section 2.1.2.

Since the introduction of the first microprocessor, all further design changes have been to improve the processor performance due to the ever-increasing demand for faster data processing. A relatively straightforward way to speed up a processor is to make transistors, which are the basic blocks of the chip, smaller and faster, enabling more transistors to be placed on the single chip and increasing the processor clock speed. One of the Intel's founders, Gordon Moore, made a prediction in 1965 that the number of transistors on the chip would double every 18 months [Moo65]. This prediction, referred to as Moore's law, has been surprisingly accurate: while Intel's 4004 microprocessor had 2300 transistors and had a clock speed of 108 kHz, today's processors may have hundreds of millions of transistors on a chip and can operate at gigahertz frequencies such as Intel's Itanium 2, for example [MN03]. Furthermore, this allows the speeding up of the processor by increasing the processor data width and by enabling the addition of full integer and floating-point arithmetic.

Research on increasing the chip density continues. However, it faces many obstacles. One of the major problems is that current designs are approaching the physical limit of semiconductors where classical physics laws are no longer applicable and quantum effects are to be considered, as shown in [Llo00] and [Fra02]. Therefore, promising technologies such as nano and molecular ones are being developed [Lun02], [BDG02]. Another key problem of current semiconductor technology is the increase in the chip power dissipation that has grown from just a few watts in the first microprocessors to more than one hundred watts for some processors such as the Intel Itanium 2 [Int03a], for example. This power is dissipated as heat and requires special cooling systems. Otherwise, the processor may become inoperable or can even be physically destroyed. Thus, a new research direction in the area of low power electronics has appeared, aimed at designing power aware

computer architectures to reduce power consumption. Major methodologies of the low power design are introduced in the book [RP96].

2.1.2 Pipelining

The previous section described those advances in semiconductor technology that allow the placing of large amounts of transistors on a chip and make it possible to explore different designs to speed up microprocessors. Burger and Goodman present speculations in [BG97] about various potential processor designs when a one billion transistor chip is available. However, the scope of this thesis is mainstream SISD processors. Therefore, the following sections present design changes to extend and speed up von-Neumann architecture microprocessors.

One of the most significant changes in processor design came from the understanding that the execution of instructions can be overlapped in time. This potential for overlapping instructions is called ILP (instruction-level parallelism). One of the first techniques to exploit ILP comes from the observation that instruction execution stages in von-Neumann architectures, described in the previous section, are potentially independent for different instructions. The technique for overlapping the execution of different stages of instructions is called pipelining [RL77]. This name appeared due to the analogy with pipelines when a continuous stream of instructions passes the processor and each part of the processor simultaneously executes different stages of different instruction in the stream. Figure 2.2 demonstrates this technique and shows the execution of two instructions on non-pipelined and pipelined processors with five abstract execution stages.

This technique was first implemented in the IBM's Stretch computer in 1959 as described in [Blo59]. However, implementing a pipeline in the first CISC microprocessors had been problematic due to the variable number of cycles for each instruction execution [HP96]. RISC architectures overcome this problem. These architectures and their advantages over CISC architectures are presented in [PD80]. Briefly, RISC architectures have a minimal instruction set consisting of the most commonly used instructions and simplified hardware that enables pipeline implementation, optimised for the fastest possible execution with a reduced number of cycles per instruction. The first implementations consisted of five main stages

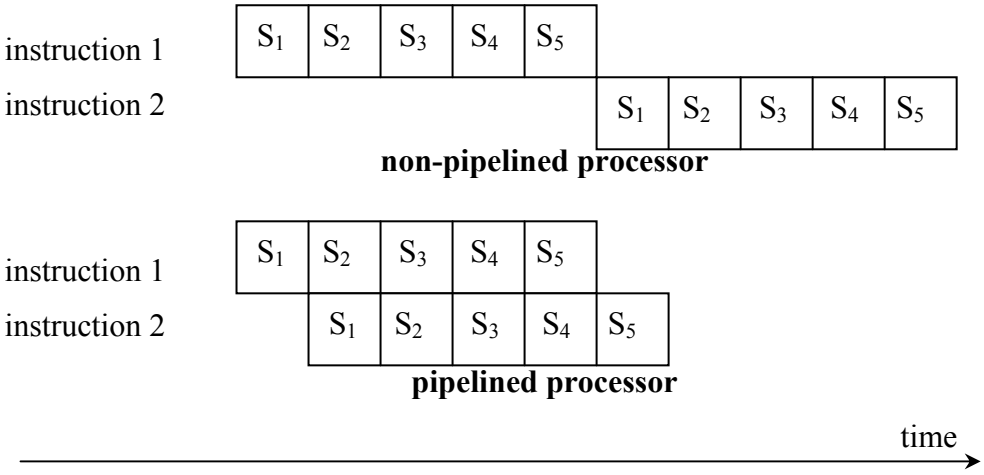


Figure 2.2: Instruction execution on non-pipelined and pipelined processors

[HP96]: fetching, decoding, instruction execution, accessing memory and writing data back to register, variations of which can still be found in most of the current processors. One of the costs for such architectures is a more complex programming target, unlike CISC processors where the complex instruction set is aimed at easing programming. This resulted in the development of special automatic compiler techniques described in section 2.3.

One of the main problems that degrade the high potential performance of pipelined processors happens when the instruction is stalled in the pipeline and so are all the following instructions. This situation is called a hazard and may occur in three cases [HP96]. The first hazard type is called data hazard and arises when one of the instructions in the pipeline depends on the results of a previous instruction. In this case, the execution of this instruction has to be delayed until the dependence is resolved. Figure 2.3 shows an example of the behaviour of the pipeline when the abstract stage S_3 of the second instruction depends on the results of the first instruction. Three possible types of data dependences exist in this hazard [RL77]. A RAW (read after write) dependence arises when an instruction attempts to read from a source before an earlier instruction writes into it so that it gets an old value. It is often referred to as a true dependence. A WAR (write after read) dependence arises when an instruction writes to a source before an earlier instruction reads it so that the earlier instruction gets a new value. A WAW (write after write) dependence arises when an instruction writes to a source before an earlier instruction writes into it so

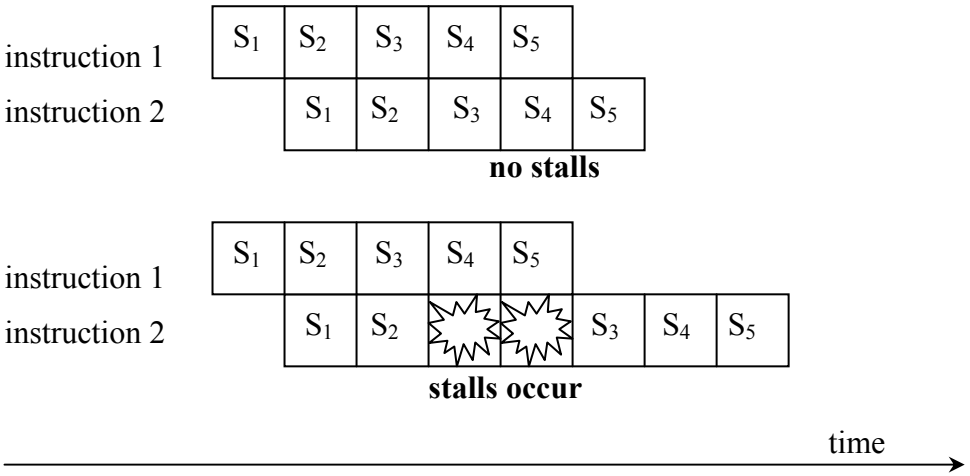


Figure 2.3: Instruction execution on a pipeline when stall occurs

that writes are performed out of order. WAR dependencies are often referred to as anti-dependencies and WAW ones as output dependencies. It is possible to minimise data hazard stalls or even eliminate some of them on a hardware level by a forwarding technique, when the result of the current instruction is forwarded immediately to all processor units that may potentially need it [HP96]. Another way to reduce data stalls is by better instruction scheduling to improve pipeline performance as shown in section 2.3.

The second hazard type is called structural hazard and occurs when the same processor functional unit such as the ALU is used in more than one stage of the pipeline and several instructions need it at the same time. This may happen when for example the ALU is used for both data and address calculations. In this case, two subsequent instructions that have some data calculations and memory access may attempt to use the ALU at the same pipeline stages that will cause a stall. However, it can be solved by duplicating the functional units to allow all possible combinations of instructions in the pipeline without structural hazard stalls and is based on the trade-off between the cost and the speed of the processor.

The last hazard type is called a control hazard and arises in branch instructions, when a decision has to be made as to whether or not to take the branch, but the information on which it is based is not yet available. One of the simplest solutions to cope with this problem is to continue fetching instructions after the branch and if the branch is taken, all the fetched instructions are cleared and the fetch is restarted from

the new address. However, if the branch is always taken, the pipeline will be always flushed after this instruction, thus, considerably degrading performance. To prevent such situations, a branch prediction table is used. It stores information about whether the particular branch was taken or not so that when this instruction is executed again, the processor will continue executing the instruction in the pipeline from the predicted address. Nevertheless, it can be potentially problematic to predict the outcome of the branch on the first occurrence or when the condition for the branch changes frequently. In this case, software methods for program analysis and branch prediction are used in cooperation with hardware methods. It is shown in section 2.3 and chapter 3.

One of the measures of how well the instructions are overlapped is CPI (clock cycles per instruction). It can be used to analyse the effectiveness of the pipeline for the particular program. Ideally, if all data and control stalls are eliminated it is possible to achieve the maximum performance of one cycle per instruction. However, further potential improvement in performance is possible by fetching more than one instruction in parallel and is discussed in the following section.

2.1.3 Superscalar processors

The CPI of the pipelined microprocessor is always limited by 1. However, it is possible to further improve performance if the microprocessor has the capability of issuing more than one instruction simultaneously and execute them in parallel. In this case, the CPI is not limited and can be far less then 1. Processors that have a pipelined architecture but are enhanced with a multiple-issue capability, are called superscalar microprocessors and described in detail in books [Joh91] and [HP96]. The straightforward design change to enable pipelined processors executing instructions in parallel would be to duplicate functional units and to add issue logic to fetch two or more instructions simultaneously. However, the main challenge in the design of the superscalar processor is to cope with those instructions that have dependencies without stalling the processor.

Instruction dependencies are classified into three types. The first type called “data dependencies” occurs when simultaneously issued instructions are data dependent and therefore cannot be executed in parallel. It corresponds to the RAW

data dependence in the processor pipeline. The second type called “name dependencies” or “storage conflicts” arises when the same registers or memory locations are used by simultaneously issued instructions. It corresponds to WAR and WAW data dependencies in the pipeline. The last type called “control dependencies” occurs when there is a branch instruction among simultaneously issued instructions.

The simplest way to cope with these hazards would be to stall the processor until all of them are resolved, however, it can considerably degrade performance. Nevertheless, it is possible to overcome data hazards by better static compiler scheduling, as shown in section 2.3 and chapter 3, or by dynamic scheduling where the processor rearranges the order in which instructions are executed. It enables the processor to look ahead of the instructions with dependence or resource conflicts and execute further independent instructions instead of stalling. This approach is implemented by inserting a buffer called an “instruction window” between the decode and execute stages. In this case, the processor places instructions into this window and then issues those instructions that do not have any dependencies and thus can be executed. This can result in the out of order issue of instructions from the buffer and therefore processors that use this approach are referred to as processors with out-of-order execution.

Overcoming WAR and WAW hazards is possible by providing additional buffers, called reservation stations, that fetch operands of the decoded instructions as soon as they are available, and by renaming the same destination registers with the names of different reservation stations. This technique, called register renaming, can therefore eliminate name dependencies between instructions.

Preventing the processor from stalling on branch instructions can be achieved by using speculation techniques, where the execution of the instructions following the branch continues even if it has not been decided whether this branch is taken. A branch-prediction buffer can assist the speculation by keeping information about whether this branch was taken or not last time. However, if the branch is wrongly predicted during out-of-order instruction execution and the program continues executing, it can generate incorrect results. Therefore, special speculation status bits are attached to instructions and registers. The use of these bits allows the processor to

mark all instructions that are executed after the branch, so that if the branch prediction failed, the results of the wrongly executed instructions can be discarded.

Finally, it should be noted, that there are ways, other than superscalar techniques, that can execute instruction in parallel, thus, utilising ILP and improving performance. There are systems consisting of multiple processors, processing vector data, using multi-threading. However, this thesis considers only the mainstream scalar processors where there are two major alternative designs to superscalar microprocessors. These are the VLIW (very long instruction word) approach [Fis83] and the EPIC (explicitly parallel instruction computing) [SR00].

Unlike superscalar processors, where the performance is improved by dynamic rescheduling of instructions, VLIW processors use a single instruction that explicitly specifies several concurrent operations independent from each other. They have a simplified hardware without dynamic scheduling or dependencies resolutions thus relying on compilers and other software methods to pack and schedule instructions. Furthermore, the code produced is generally not portable across different architectures and thus, is not used for general-purpose computing. However, it is popular in DSP (digital signal processing) applications where most of the execution time is spent in small kernels that are relatively easy to analyse for dependencies and to optimise on the assembler level for the specific DSP processor.

The EPIC approach combines some features of VLIW and superscalar processors. It relies on the compiler to extract instruction level parallelism and to schedule independent instructions statically as in the case of VLIW. However, it is also similar to a scalar processor with a sequential instruction set that allows programs to be portable among various processor implementations. This approach is used in Intel's IA-64 processors, as described in [HMR+00], [MN03] and [Int03a].

2.2 Memory hierarchy

The previous section concentrated on how to improve the processor performance. However, the overall computer performance depends not only on the processor speed but also on the speed of all components. One such component is the memory system. This section describes memory design evolution, locality and cache classifications.

2.2.1 Memory design evolution

Computer memory is used as storage for program code and data. It is one of the key computer components and influences the overall computer performance by taking the burden of supplying data steadily to the microprocessor. The simplest design of the computer system would be if all data is kept and processed in the same non-volatile memory. In practice, however, it is not feasible because permanent memory devices such as tapes or magnetic and optical disks are generally slow. The economical and electronic trade-off in this technology is that it is possible to build fast but small or large but slow memory systems. Therefore, a memory hierarchy, based on speed, size and cost is used. It was first introduced in the Atlas computer that was developed at the University of Manchester [KE62].

A typical memory hierarchy contains registers inside the processor, which are small in number but provide immediate access; reasonably fast and large main memory for storing both program code and data; and finally some slow permanent storage. Scheible surveys various hardware memory designs in [Sch02]. Main memory is often referred to as RAM (random access memory) because any word in such memory can be accessed in random order. There are two basic types of RAM: SRAM (static random access memory) and DRAM (dynamic random access memory). The difference between these two types is in the hardware implementation. Dynamic memory has a simple design and needs to be refreshed periodically so as not to loose data, thus, making this memory cheap but relatively slow. Static memory has a more complex design without the need to be refreshed, thus working faster than DRAM. However, it is physically larger and more expensive. Therefore, DRAM is a common choice for main memory.

Advances in semiconductor technology improve the main memory size and transfer speed by placing more transistors with higher density on the chip. Furthermore, it is also possible to improve DRAM performance by changing memory and interface design. Some of those designs are surveyed in [Sch02]. Briefly, one of the techniques is to make the bus that connects the processor and memory wider so that more data can be sent to the processor within each cycle. Another technique, called interleaving, is based on separating memory into several independent banks and allowing access to multiple data at the same time without

conflicts. Synchronous DRAM (SDRAM) can speed up sequential memory access by matrix interconnection topology. Finally, two recent competing technologies are Rambus DRAM (RDRAM) and Double Data Rate DRAM (DDR DRAM). RDRAM provides a new interface with a packet-based protocol that allows overlapped memory transactions. DDR DRAM uses a technique where data is transferred between processor and memory on both the rising and the falling edges, thus, doubling memory speed without any increase in clock frequency [CJD+01].

2.2.2 Locality and cache design

The previous section described advances in the hardware design of main memory to improve its speed. Nevertheless, the gap between processor and memory performance is widening exponentially [HP96]. One of the most commonly used techniques to solve this problem is based on placing small and fast intermediate storage between the main memory and the registers within the processor [Smi82]. This small storage is called cache memory and is used to keep frequently used data and code closer to the processor so that it can access them faster.

Cache memory exploits locality. There are two types of locality – spatial and temporal. Spatial locality means that a memory location adjacent to a referenced location is likely to be referenced soon. Temporal locality means that a memory location recently referenced is likely to be referenced in the nearest future. Whenever the processor requests an item of data from memory, it first checks whether this data can be found in cache. If data is not in cache, a cache miss occurs. In this case, data is fetched from slow main memory to the processor and is simultaneously placed into cache. If the program exhibits temporal locality so that the processor requests the same data later, a cache hit occurs and this data is only fetched from the fast cache, thus, considerably reducing the overall memory access time. To exploit spatial locality, a fixed-size block of adjacent data to the requested data is also fetched from main memory to cache on a cache miss. Therefore, if the processor requests this adjacent data later, it is fetched directly from the cache, speeding up execution of the program.

When data is moved to the cache, the location within the cache is determined by its original address and the cache organisation. The cheapest and simplest

organisation is used in direct-mapped caches, where each memory location can be mapped to one unique location in cache using the modulo function:

$$Address_{cache} = Address_{main\ memory} \quad MOD \quad Size_{cache}$$

However, the major drawback of this cache type is the reduced capability for exploiting locality. This happens when new data is fetched to an already allocated place in cache, so that old data has to be replaced even if it can be potentially used in the near future. In contrast, fully associative caches allow data to be placed anywhere in cache. However, this cache organisation is more complex and expensive as the cache now keeps not only the data but also its corresponding address. It needs to have fast logic for finding this data by comparing the requested address with all the stored addresses in the cache simultaneously (associatively). Therefore, due to economical reasons set-associative caches are used. They consist of a number of sets so that the memory location is first mapped to the set using a module function in the same way as direct-mapped caches. Data can then be placed anywhere within the set. If there are *n* possible locations in the set where data can be placed, the cache is called *n*-way set associative. When the set is full, some data should be replaced. Two most commonly used replacement strategies are random, where data is replaced randomly within the set and LRU (least-recently used) strategy, where data accesses are recorded and the least used data is replaced, thus, attempting to exploit temporal locality.

The above methods are used to speed up data reads from memory because in practice they dominate memory access. However, memory writes can also considerably degrade the overall performance. There are two main cache policies for cache behaviour when a data write occurs. The simplest policy is “write through” where data is written to both cache and main memory, however, it usually stalls the processor until the operation is finished. One solution is not to stall the processor by introducing a write buffer that allows overlapping processor execution with writing to memory. Another policy is called “write back”. This policy allows data to be written back to the cache without writing it to memory first. Only when this data is to be replaced in cache due to other memory requests, is it written to main memory. This policy better exploits temporal locality, but the cache organisation is more complex and needs to control data synchronisation between cache and main memory.

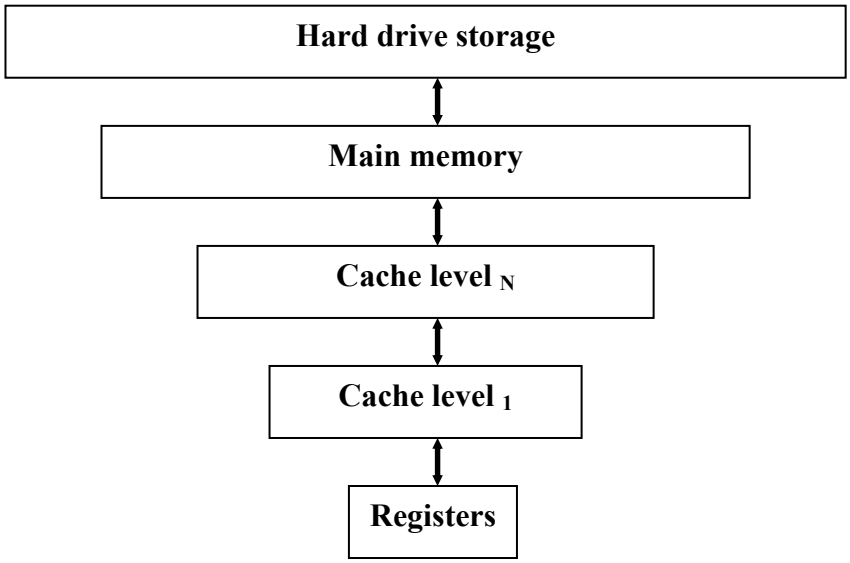


Figure 2.4: Memory hierarchy in current computing systems

Since the gap between processor and memory speed continues to grow and the amount of data to process grows as well, the memory hierarchy continues to alter as shown in figure 2.4. Multilevel caches are introduced to accommodate the increasing gap between the speed of the main memory and the cache, so that the closer the cache is to the processor the smaller and faster it is.

Virtual memory allows the processing of larger amounts of data than the main memory size. It uses main memory as a cache for larger storage such as the hard drive by dividing it into pages so that when the memory access occurs it is mapped to a specific page. If a page is not in memory, a page fault occurs and this page is loaded into the main memory from the hard drive or other storage. Since the cost of a page fault is high due to the access to the relatively slow devices, a fully associative policy is usually used so that pages can be placed anywhere in main memory. When main memory is full, the least recently used page is replaced. To speed up the mapping of physical addresses to virtual addresses, a page table or TLB (translation look-aside buffer) is used. It caches the physical addresses of recently used pages and, thus, provides a fast translation from virtual to physical addresses.

An important characteristic that shows the cost of memory behaviour is the cache miss rate, which is the percentage of the memory accesses that result in cache misses. There are three sources of misses: compulsory, capacity and conflict. Compulsory

misses occur when data is brought to the cache for the very first time. Capacity misses occur due to the limited cache size so that when it is full, some data that is still in use nevertheless has to be replaced. Conflict misses occur in direct-mapped or set-associative caches when too many main memory lines are mapped to the same cache set so that some data, later accessed, has to be discarded.

Reducing cache misses means reducing the number of costly accesses to main memory and therefore potentially speeding up the program execution. Many different techniques have been proposed to reduce data traffic between main memory and the cache. Software methods to analyse program behaviour and reduce cache misses are surveyed in chapter 3. Hardware methods are out of the scope of this thesis and the most common of them are described in [HP96]. Briefly, they are based on making the block size larger to bring more data from main memory on the cache miss thus, reducing compulsory misses. Reducing conflict misses is possible by increasing the associativity of the cache or by using different designs such as column-associative caches [AP93], skewed-associative caches [BS95], victim caches or by using randomised cache placement [TG99]. However, it should be noted that while reducing cache miss rate improves the performance of the in-order processors, where each cache miss causes the stall, it does not necessarily improve the performance of current out-of-order execution processors due to the potential overlapping of memory access with executing other instructions instead of stalling. This is examined in detail in chapter 5.

2.3 Compiler technology

The compiler is an important software component of any computing system, responsible for translating user program into machine code. This section contains a brief survey of compiler technologies and describes major compiler optimisations to produce faster code for superscalar processors with out-of-order execution, excluding memory optimisations, which are described in chapter 3.

2.3.1 Introduction to compiling

Early computers were programmed directly using binary machine code. However, this process is not only tedious and time consuming, but also requires a good knowledge of the underlying computer hardware. Moreover, binary code is difficult to analyse and modify if any further changes are necessary. Furthermore, such codes are generally not portable to new architectures. Therefore, an assembly language is used instead. It translates program source code containing computer instructions into machine code. The assembler usually has some basic support for data structures and subroutines making it easier to develop and modify programs. However, it still requires knowledge of the particular architecture instruction set and is not portable between different platforms.

This problem has been solved by introducing high-level computer languages and their respective compilers. High-level languages are usually designed for some particular classes of problems and are platform independent allowing programmer to write compact portable programs. Compilers, however, are typically platform dependent and translate programs written on the high-level language into the assembly language or machine code of the targeted architecture. One of the earliest languages and compilers developed for scientific applications was Fortran. It is not only still in use today, but it also became a standard for numerical programs [PTV+92]. Fortran compilers use mature technology that has been developed over many years and is now capable of producing high quality fast code for a variety of platforms.

Compilers transform source code into machine code through different stages. The common stages are lexical, syntax and semantic analysis that form the compiler front-end. Code optimisation and machine code generation constitute the compiler back-end. These stages are described in detail in [ASU86].

Briefly, the compiler front-end is responsible for checking that the program is correct lexically, syntactically and semantically. It constructs an abstract intermediate representation of the program. This intermediate representation removes redundancies in the application and contains only unique machine-independent information about the original program. This simplifies the retargeting of the compiler for different languages and platforms so that only the compiler front-end is

changed for a new language and the compiler backend is changed for a new platform. The code optimisation stage remains intact. The code optimisation stage is responsible for improving the quality of the intermediate code so that faster and smaller machine code will be produced. This stage will be described in more detail in section 2.3.2 and in chapter 3. Finally, the compiler back-end is responsible for producing assembly or machine code from the program intermediate representation for the target platform. During this stage, registers are allocated and certain platform-specific optimisations, such as instruction scheduling, are performed. It is described in the next section.

2.3.2 Code optimisations

Using high-level languages helps the programmer to abstract from the underlying machine architecture, to have an easier and simpler development process and to write compact portable programs. However, this means that the compiler has a major role in producing fast and efficient target machine code automatically. This is not a trivial task because potentially many variants of the machine code exist for the same program. Hence, the task of the compiler is to find and produce the best version of the machine code for any given program. This process is called program optimisation.

Program optimisations are performed via program transformations that can improve speed and/or size of the final machine code without changing the behaviour and meaning of the program [ASU86]. These transformations are applied at different compiler stages and can either be platform independent, when properties of the targeted machine are not taken into consideration, or platform dependent when various platform parameters are taken into account.

Before optimising any program, the compiler has to perform control flow analysis and data flow analysis. Control flow analysis is usually performed in the front-end of the compiler where the intermediate representation of the program is generated. It divides the whole program into basic blocks that have only one entrance and one exit, and produces a control flow graph that shows how the basic blocks are interconnected. This helps the compiler identify loop structures and other parts of the program that can be further legally transformed.

Data flow analysis is performed on the intermediate representation of the program. It examines the flow of data in the whole program, producing information about each variable, such as where this variable is first defined, how it is used in basic blocks and finally where it is redefined. This is a complex process that requires examining all control paths of the program but simplifies further data dependence analysis and program optimisation.

Once the control and data flow graphs are available, the compiler starts optimising the program. First, machine independent optimisations are performed. These include transformations such as code motion, code inlining, common subexpression elimination and copy propagation transformations [ASU86]. Briefly, the code motion transformation moves invariant statements within a loop outside, thus, eliminating redundant computation and speeding up the overall execution of the program. Code inlining is used to remove the call statement overhead by merging small and frequently called subroutines with the caller. This transformation can speed up the program but it can also increase the size of the program if there is more than one place where the subroutine is called. Finally, both global common subexpression elimination and copy propagation are used to avoid repetitive computations, thus, improving code performance as well as code size.

The final compiler stage is to allocate register and memory resources to the program and to generate and schedule machine instructions from the program intermediate representation. It is not a trivial task, as the compiler has to take into account various machine parameters in order to produce the fastest possible code for the particular architecture. This stage is beyond the scope of this thesis and is described in detail in [ASU86] and [GH88].

Briefly, during the register allocation phase, the compiler has to determine which values should be placed in registers based on the data flow and dependence analysis. The major difficulty of this task is that there is a limited number of hardware registers. The main objective is to reduce the number of memory accesses giving a potential performance improvement. Various methods for register allocation are presented in [ASU86] and [Tou02].

During the instruction scheduling phase, the compiler has to produce and optimise the instruction sequence in such a way that data and control dependencies

are not violated and that the program's ILP is exploited without introducing processor stalls [SCD+97]. Though most modern processors have an automatic support for deriving program ILP at run time, it is limited because the processor can analyse ahead only those instructions that reside in the instruction window at a time. Compilers have the advantage of analysing the whole program and scheduling instructions for the pipeline globally and in some cases predicting branches using static information from the data flow and dependencies analysis. Furthermore, to exploit features of modern superscalar processors with out-of-order execution the compiler can perform various machine dependent optimisations.

Loop unrolling and software pipelining are two major transformations that can improve scheduling of the program and better exploit ILP. Loop unrolling replicates the loop body multiple times, thus, reducing the number of loop branch checks. This is one of the transformations used in the research for this thesis and thus, is reviewed in more detail in chapter 3. Software pipelining transforms a loop in such a way that each instruction of the new loop is assembled from instructions belonging to different iterations of the original loop, thus, allowing the overlapping of multiple instructions without data dependencies. It is described in detail in [BGS94].

One of the difficulties compilers face in this phase is the lack of precise information about the hardware of the targeted machine. Hence, simplified machine models are used that reduce the potential for exploiting ILP. This is discussed in detail in chapter 3.

2.4 Summary

The evolution of the processor design is surveyed in this chapter starting from the description of the internal structure of the world's first microprocessor, the Intel 4004. Advances in the semiconductor technology allowing higher transistor density per chip are discussed and followed by the brief analysis of various design changes to improve the processor performance. CISC and RISC processor architectures are compared and pipelining technique implementation for both architectures are discussed. Multiple issue techniques and out-of-order instruction execution for exploiting instruction level parallelism are presented. This is followed by the discussion of various hazards on superscalar processors that can considerably

degrade the performance and the hardware solutions used to overcome them. The need for memory hierarchy to accommodate the widening gap between the speed of the processor and the memory is outlined. The main memory designs are briefly surveyed, which is then followed by the introduction of caches that exploit locality to further reduce memory access time. The most common cache organisations are analysed. Finally, this chapter finishes with the introduction to the compiling technology and with the description of basic optimisations excluding memory analysis and optimisations that are discussed in the next chapter.

Chapter 3

Memory hierarchy optimisations

This chapter surveys existing work related to the research of this thesis. It reviews various program transformations that can improve memory performance by reducing data traffic between the processor and the memory and primarily focuses on loop tiling, array padding and loop unrolling, though other transformations are also briefly reviewed. It discusses certain program representations and issues concerning the legality of transformations. Various static techniques for analysing data reuse and locality and for obtaining the number of cache misses within a program are further presented. It is followed by a survey of static algorithms for transforming programs to improve data locality and reduce conflict and compulsory misses. A review of program dynamic analysis is then presented. It obtains various run-time parameters that are not available statically by means of profiling or simulations. This run-time information can be used during dynamic optimisations such as in feedback assisted, iterative or adaptive compilation, as described in the last section of this chapter.

3.1 Program transformations

This section describes various transformations and their effect on program performance. This includes major loop and data transformation used in the research of this thesis such as loop tiling, array padding and loop unrolling. Other transformations such as software pipelining and prefetching are also briefly described. Examples of mathematical notations for the representation of loops and arrays to ease dependence analysis and automatic application of those transformations are presented.

3.1.1 Introduction

The aim of new hardware designs of computing systems that target numerical codes is faster execution of a broad range of programs. Those programs are generally

developed for older platforms that do not reflect new design features, and are ported to a new hardware without major changes due to economical reasons. Therefore, there are two major ways to improve the performance of the unchanged program: by analysing and rescheduling the stream of instructions dynamically by the processor or by analysing and transforming the program statically by the compiler. The first way of scheduling instructions by the processor is limited as the processor can only analyse several instructions at a time and does not see the whole program behaviour as discussed in section 2.3.2. On the contrary, a compiler can analyse the whole program and adapt it to a new computing system using program transformations, even if the original algorithm is unchanged.

The aim of program transformations is to reorder operations in a program to improve performance without changing the meaning of the program. Program transformations that remove redundancies and improve scheduling were first introduced in this thesis in section 2.3.2. This section focuses on memory transformations that are used to overcome the increasing gap between the speed of the processor and the main memory, by improving data locality and minimising the number of non-local memory accesses.

Memory transformations are divided into two groups: loop and data transformations. Loop transformations modify loop iteration order in an attempt to achieve better data locality, without changing the meaning of the original program. The emphasis on loop structures is due to an observation that programs spend most of their execution time in loops. Data transformations modify the layout of array data in the memory with the same aim of achieving better locality.

Bacon et al. [BGS94] survey various loop and data transformations and describe their influence on program locality and performance. This thesis focuses on three transformations: loop tiling, unrolling and array padding. These transformations have been selected due to their potential to considerably improve performance. They are described in detail in the following sections 3.1.2, 3.1.3 and 3.1.4.

Transformations such as loop unrolling and array padding are relatively easy to analyse and implement, while others, such as loop tiling, require thorough dependence analysis and can be difficult to implement inside the automatic

optimising compiler. Therefore, linear algebraic models to represent loop and data structures are used to enable automatic optimisations.

One of the first significant papers that uses a mathematical model to unify various transformations is [WL91b] by Wolf and Lam. It describes a matrix model for transformations and incorporates dependence vectors to check the validity of transformations within the same framework. It provides the theory for the automatic analysis of the validity of standalone or even compound transformations and enables the data locality analysis, as is shown in further sections of this chapter. However, the loop transformation theory of this paper is limited by unimodular transformations such as loop interchange, reversal and skewing for the perfectly nested loops.

Briefly, this model represents a loop nest as a finite convex polyhedron and each iteration in this loop nest is described as an index vector $\vec{\mathbf{p}}$. Data dependencies restrain the execution order of the iterations and can be represented as dependence vectors $\vec{\mathbf{d}}$. Loop transformations map original iterations into new ones and can be represented as matrices $\mathbf{T}$. When loop transformation is applied, new iteration and dependence vector is found as the matrix-vector multiplication, that is $\vec{\mathbf{p}}_{new} = \mathbf{T}\vec{\mathbf{p}}$, $\vec{\mathbf{d}}_{new} = \mathbf{T}\vec{\mathbf{d}}$. If a compound transformation, consisting of several transformations $\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_N$ has to be applied, the final matrix of the compound transformation is found as a consecutive multiplication of all matrices $\mathbf{T}_{new} = \mathbf{T}_1\mathbf{T}_2...\mathbf{T}_N$. Finally, a new transformation is legal if the transformed code can be executed sequentially, or in mathematical terms, if all transformed dependence vectors are lexicographically positive. The definition of a lexicographically positive vector $\vec{\mathbf{d}}$ is the following: if $\exists i : (d_i < 0 \text{ and } \forall j < i : d_j \geq 0)$.

The following example, taken from [WL91b], demonstrates the use of the loop transformation theory:

```
for I1: = 1 to N do
  for I2: = 1 to N do
    a[I1, I2] := f( a[I1, I2], a[I1+1, I2-1] );
```

The dependence vector of this double nested loop is (1,-1) and the iteration vector is (i, j). Consider that in the first case the loop interchange transformation has to be applied and in the second case a compound transformation consisting of the loop

interchange and the loop reversal transformations has to be applied. The matrix of the loop interchange transformation is $\mathbf{T} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ and the matrix of the loop reversal transformation is $\mathbf{T} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$. When loop interchange is applied, the new dependence vector $\mathbf{d}_{new} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$ is lexicographically negative. Therefore, this transformation is not legal for this code. However, when the compound transformation of the loop interchange and the loop reversal is applied, the new dependence vector $\mathbf{d}_{new} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ is lexicographically positive. Hence, this compound transformation can be legally applied to the above code.

Analysing data dependencies for large programs can be a complex and slow process and can be hard to implement inside a production compiler. The Banerjee test [Ban88], based on the Intermediate Value Theorem, is commonly used to detect all the dependencies between variables within a given region of the program. A faster method for determining data dependence relationships based on an integer programming algorithm is introduced by Pugh in [Pug91]. To speed up the analysis, an Omega test is used. This method allows one to determine if an integer solution exists to a set of linear equalities and inequalities.

Another important issue, which arises after applying some particular transformations, such as loop tiling, is that loop bounds have to be transformed as well. Calculating new loop bounds can be performed directly by transforming all inequalities derived from the loop nest, as proposed originally by Wolf and Lam in [WL91b]. However, it may potentially contain excessive maxima and minima computations in the new loop bounds that can degrade overall performance. To overcome this problem, Ancourt and Irigoin introduce several algorithms in [AI91] that optimise minima and maxima computations in the loop bounds using integer linear system methods.

Li and Pingali extend unimodular transformation theory in [LP92] by using non-singular matrices for transformations. In this loop transformation framework called

do $x_1 = L_1, U_1$
S_{1a} : $H_{1a}(x_1)$
do $x_2 = L_2, U_2$
S_{2a} : $H_{2a}(x_1, x_2)$
. . .
do $x_n = L_n, U_n$
S_n : $H_n(x_1, \ldots, x_n)$
. . .
S_{2b} : $H_{2b}(x_1, x_2)$
S_{1b} : $H_{1b}(x_1)$

(original loop)

do $x_1 = L_1, U_1$
do $x_2 = L_2, U_2$
. . .
do $x_n = L_n, U_n$
S_{1a} : **if** $x_2 = L_2 \wedge \ldots \wedge x_n = L_n$ **then** $H_{1a}(x_1)$
S_{2a} : **if** $x_3 = L_3 \wedge \ldots \wedge x_n = L_n$ **then** $H_{2a}(x_1, x_2)$
. . .
S_n : $H_n(x_1, \ldots, x_n)$
. . .
S_{2b} : **if** $x_3 = U_3 \wedge \ldots \wedge x_n = U_n$ **then** $H_{2b}(x_1, x_2)$
S_{1b} : **if** $x_2 = U_2 \wedge \ldots \wedge x_n = U_n$ **then** $H_{1b}(x_1)$

(transformed loop)

Figure 3.1: Abu-Sufah's transformation of imperfectly nested loop to a perfect loop nest.

Λ-transformations, some new transformations can be used, in addition to all unimodular transformations that are included as a sub-case. One such transformation is loop skewing, which non-singular transformation matrix is $\mathbf{T} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$. In addition, integer lattice theory is used to generate efficient code. This paper also contains a proof that any transformation, which can be represented by an integer non-singular matrix, can be composed using four basic transformations: permutation, skewing, reversal and scaling.

To expand the loop transformation theory on non-perfectly nested loops, Xue suggests converting an imperfectly nested loop to a perfect loop nest in [Xue97a] by using Abu-Sufah's Non-Basic-to-Basic-Loop transformation. This allows one to apply unimodular transformations and extract data dependencies in the usual manner. Figure 3.1 demonstrates an example from this paper for transforming an n-deep non-

perfectly nested loop to a perfectly nested loop. This approach has two major drawbacks. The first one is that the Non-Basic-to-Basic-Loop transformation is not always legal and the issue of legality is discussed in this paper. The second one is that the innermost loop contains an excessive amount of "if" statements that can degrade performance considerably, particularly on pipelined processors with out-of-order execution, as discussed in sections 2.1.2 and 2.1.3.

To unify various loop transformations and to extend their applicability to arbitrary loop nests, affine partitioning was proposed by Lim and Lam in [LL97]. Originally, this paper suggested using affine partitions to maximise parallelism and minimise synchronisation for multiprocessor computing systems. Later, this method was extended to optimise data locality for uniprocessors in [LLL01] by Lim et al. Briefly, this model uniquely identifies all operations by the loop index values of the enclosing loops. It expresses all possible combinations of various transformations using affine transforms that are created for each operation to map old index values into new ones. Depending on the task, various search algorithms are used to find the optimal affine transform for maximising parallelism or improving data locality. Feautrier provides some additional details about solving affine scheduling efficiently in [Fea92].

Besides loop transformations, data transformations can also benefit from the mathematical representation. O'Boyle and Knijnenburg introduce a single framework in [OK99] that unifies various non-singular data transformations. It allows one to perform compound transformations in one step, using matrix representation for arrays and transformations in a similar way to the framework proposed by Wolf and Lam in [WL91b]. For example, if $\mathbf{J}$ is the iteration vector and there is an access to an array A(i+j, j) inside a two-nested loop, then the subscripts of this array can be written as an affine mapping $\mathbf{UJ} + \mathbf{u} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. Data transformation in this model consists of a non-singular matrix $\mathbf{A}$ and a shift vector $\mathbf{a}$ and its application to the array access results in a new access, where $\mathbf{U'} = \mathbf{AU}$ and $\mathbf{u'} = \mathbf{Au} + \mathbf{a}$. It should be noted that data transformations affect all accesses to the particular array globally, unlike loop transformations that are local for the loop nest.

do i = 1, n
  do j = 1, n
    A(2*i,i+j) = i+j

$$\begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+\begin{bmatrix} 0 \\ 0 \end{bmatrix}=\begin{bmatrix} 2i \\ i+j \end{bmatrix} \rightarrow \mathbf{U}=\begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix},\mathbf{u}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

do i = 1, n
  do j = 1, n
    A(2*i,j) = B(i+j)

$$\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix}+\begin{bmatrix} 0 \\ 0 \end{bmatrix}=\begin{bmatrix} 2i \\ j \end{bmatrix} \rightarrow \mathbf{U}=\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix},\mathbf{u}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

(original access to array A)

$$\mathbf{A}=\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix},\ \mathbf{a}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\mathbf{U'}=\mathbf{AU}=\begin{bmatrix} 2 & 0 \\ 3 & 1 \end{bmatrix},\mathbf{u'}=\mathbf{Au}+\mathbf{a}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\mathbf{U'}=\mathbf{AU}=\begin{bmatrix} 2 & 0 \\ 2 & 1 \end{bmatrix},\mathbf{u'}=\mathbf{Au}+\mathbf{a}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

do i = 1, n
  do j = 1, n
    A(2*i,3*i+j) = i+j
do i = 1, n
  do j = 1, n
  A(2*i,2*i+j) = B(i+j)

(transformed access to array A after array skewing)

Figure 3.2: Data transformation theory examples

Figure 3.2 demonstrates data skewing transformations for the array A and the respective transformation matrices and shift vectors. The advantage of this framework is that it unifies and eases the analysis and application of data transformations, and can be easily implemented inside production compilers.

Finally, an approach to combine loop and data transformations that can achieve better results than if those transformations are used separately, is presented by Kandemir et al. in [KCR+98]. This paper proposes an integrated compiler framework that combines both loop and data transformations for optimising data locality for numerical codes. The following sections contain further information about three transformations that are used in the research of this thesis: loop tiling, array padding and loop unrolling.

3.1.2 Loop tiling

Loop tiling (blocking) is a transformation that is used to improve cache reuse within a loop nest, by dividing the iteration space of the nest into fixed-size blocks.

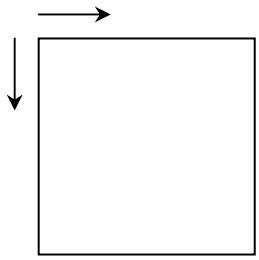
original loop nest:

```
do I = 1, N
    do J = 1, N
        A(I,J) = A(I,J) + B(I,J)
        C(I,J) = A(I-1,J) * 2
    end do
end do
```

transformed loop nest:

```
do IT = 1, N, SS
    do JT = 1, N, SS
        do I = IT, MIN(N, IT+SS-1)
            do J = JT, MIN(N, JT+SS-1)
                A(I,J) = A(I,J) + B(I,J)
                C(I,J) = A(I-1,J) * 2
            end do
        end do
    end do
end do
```

iteration space of the original loop:



iteration space of the transformed loop:

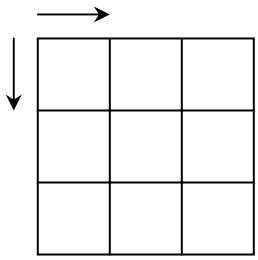


Figure 3.3: Loop tiling example

This transformation can be used in cases where the data footprint of the original loop nest, which is defined as the amount of data touched within this loop nest, is bigger than the cache size. In this situation, if the tile size is chosen to fit the new data footprint into cache, it can result in better data reuse inside the new loop structure.

Wolfe describes various practical examples of loop tiling and discusses issues of the legality of this transformation in [Wol89]. A simple example of tiling a 2-nested loop from this paper is shown in figure 3.3. It demonstrates how the iteration space is divided into blocks of size SS*SS to improve data locality.

Xue uses a mathematical formulation similar to the one introduced in section 3.1.1 to analyse the effects of the tiling transformation on data dependencies and to ease the dependence test for the legality of the transformation [Xue97b]. Figure 3.4 presents a generalised version of the loop tiling for the m-dimensional loop nest with tile factor of B. Loop tiling decomposes this m-dimensional loop nest into 2m-

original loop nest:

```
do i1 = 1, N
  do i2 = 1, N
    …
      do im = 1, N
        S(i1, i2, …, im)
```

transformed loop nest:

```
do ii1 = 1, N, B
  do ii2 = 1, N, B
    …
      do iim = 1, N, B
        do i1 = ii1, N, min(N, ii1+B-1)
          do i2 = ii2, N, min(N, ii2+B-1)
            …
              do im = iim, N, min(N, ii3+B-1)
                S(i1, i2, …, im)
```

Figure 3.4: Generalised version of loop tiling

dimensional loop nest, so that the innermost m loops iterate within a block, thus, improving data locality.

Finally, it should be noted that it is also possible to tile imperfectly nested loops. More information about the tiling of imperfectly nested loops can be found in paper [AMP00].

The major question before applying loop tiling transformation is how to choose the tile size to improve the performance of the code. Data locality analysis and analysis of conflict cache misses are used to derive this information and are discussed in section 3.2.

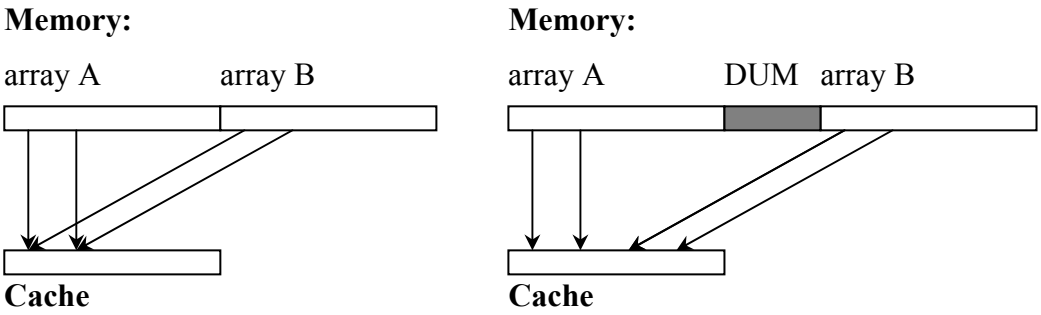
3.1.3 Array padding

Array padding modifies the program data layout to remove cache conflict misses that occur due to a limited cache set associativity, as briefly described in section 2.2.2. This transformation has two types: inter- and intra-variable padding. Inter-variable padding changes the base addresses of arrays, and intra-variable padding inserts dummy data locations between the columns of arrays. Rivera and Tseng describe both types and present an analysis and heuristic to apply this transformation in [RT98].

real S, A(N), B(N) ⟶ real S, A(N), DUM(PAD), B(N)

do i = 1, N

  S = S + A(i)*B(i)



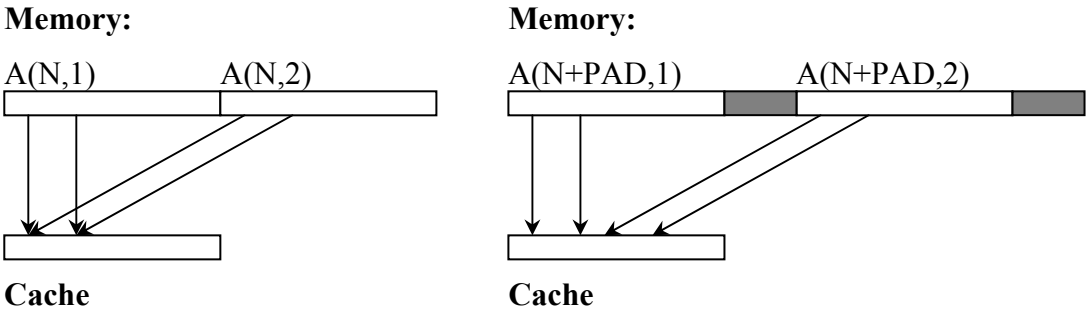
(a) Inter-variable padding

real A(N,N), B(N,N) ⟶ real A(N+PAD,N), B(N,N)

do i = 2, N-1

  do j = 2, N-1

    B(j,i) = (A(j-1,i)+A(j,i-1)+A(j+1,i)+A(j,i+1))/4



(b) Intra-variable padding

Figure 3.5: Intra-variable and inter-variable array padding examples

Figure 3.5 (a) presents an example for inter-variable padding from this paper. Unit-stride consecutive accesses to arrays A and B have a potential for spatial locality. However, if the cache is direct-mapped and if these arrays are situated in memory at such addresses that every access to A(i) and B(i) is mapped to the same cache line, then every reference will generate a cache conflict miss, thus, degrading performance. To solve this problem and ensure cache reuse, inter-variable array padding is used. It changes the base address of the array B in such a way that references to A(i) and B(i) are mapped to different cache locations. The array base

original loop:

```
do i = 2, n – 1
  a[i] = a[i] + a[i-1] * a[i+1]
end do
```

loop unrolled twice:

```
do i = 2, n-2, 2
  a[i] = a[i] + a[i-1] * a[i +1]
  a[i+1] = a[I+1] + a[i] * a[i+2]
end do
if (mod(n-2,2) = 1) then
  a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

Figure 3.6: Loop unrolling example

address can be changed directly on assembler level, or indirectly on source level, by inserting some dummy array DUM between arrays A and B.

Similar situations may occur in cases of multidimensional arrays. Figure 3.5(b) shows an example of stencil computation from the same paper [RT98]. Memory references to the 2-dimensional array A in this example have a potential for spatial and temporal reuse. However, if the column size of array A is a multiple of the cache size, all columns of this array will map to the same cache lines and will generate cache misses. Therefore, inter-variable padding is applied by inserting some dummy locations between array columns to avoid their mapping to the same cache lines. Finally, a generalised version of intra-padding with a PAD factor for array A(N_1, N_2, …), as used in the research of this thesis, is A(N_1+PAD, N_2, …).

The major questions before applying array padding are how to detect conflicting array references and how to choose the size of the dummy array to reduce conflict misses. The data layout analysis and optimisations are presented in section 3.2.

3.1.4 Loop unrolling

Loop unrolling is used primarily to improve instruction level parallelism and reduce loop overhead by replicating the body of the loop a number of times and replacing the loop step with this number. In addition, loop unrolling can also improve data locality and register usage by reducing the number of memory accesses and is therefore within the scope of memory optimisations. Dongarra and Hinds briefly describe this transformation and show its effect for various unrolling factors on two

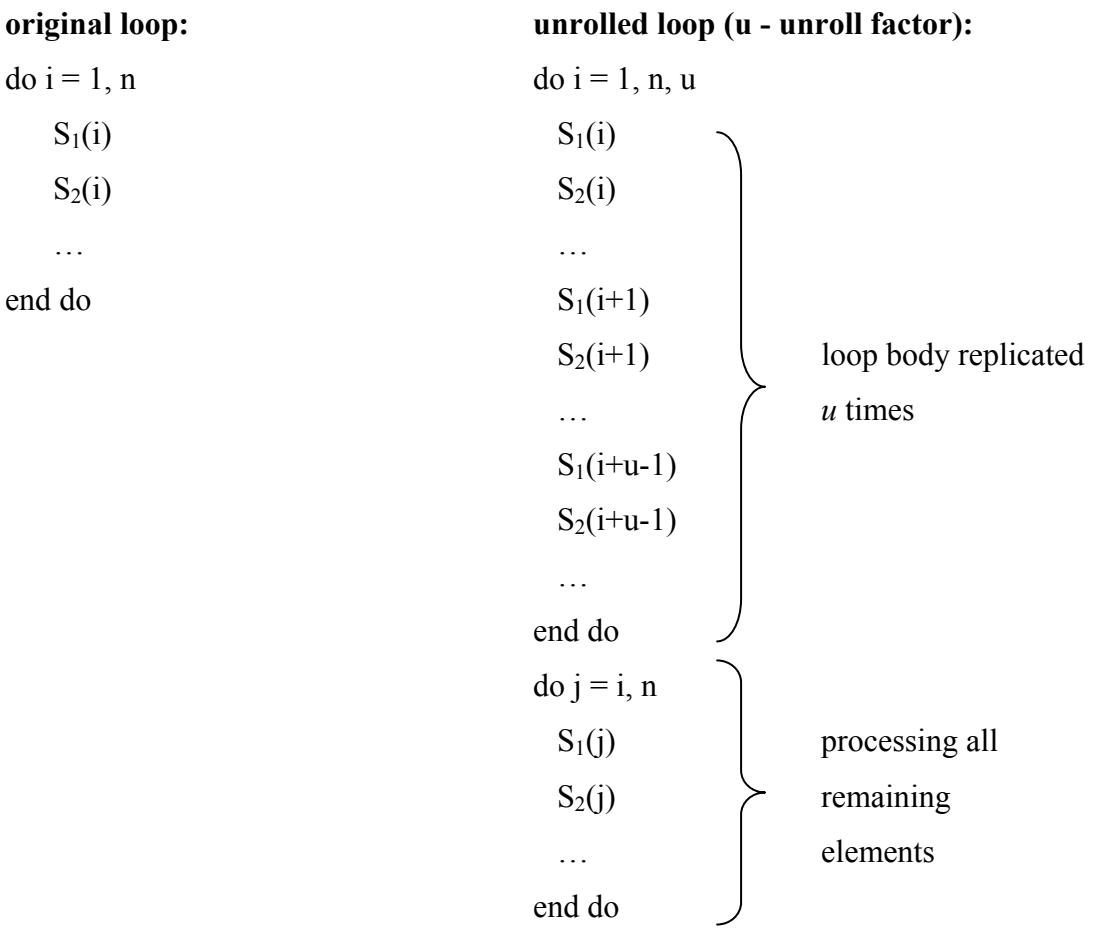


Figure 3.7: Generalised version of loop unrolling

subroutines in [DH79]. Bacon et al. analyse loop unrolling in more detail in [BGS94]. Figure 3.6 presents an example from this paper and shows an original sample loop and unrolled loop with a factor of two.

This example demonstrates that the loop overhead that consists of the increment, test and branch operations, is 2 times less than in the original loop. Moreover, reducing the number of branches can reduce the number of control dependencies and thus improves program performance on modern pipelined processors with out-of-order execution, as discussed in sections 2.1.2 and 2.1.3. This example also demonstrates how data or register locality can be improved: array references A[i] and A[i+1] are used twice in the unrolled loop, thus, reducing the number of memory accesses from 3 to 2 per iteration. It should also be noted that the *if* statement at the end of the unrolled loop, in this example, is needed when it is not known at a compile time whether the total number of iterations in the loop is a multiple of the unrolling factor or not. If it is not an exact multiple then this code is needed to process the

remaining elements. However, it is also possible to generalise loop unrolling using two loops, as shown in figure 3.7, where an additional loop is needed to process all remaining elements.

Finally, it should be noted that applying unrolling is always legal for a single loop. However, the major question is how to choose the best unrolling factor to improve performance. This is discussed in section 3.2.1.

3.1.5 Other transformations

There are many other transformations besides loop tiling, array padding and loop unrolling that can potentially improve program performance. Many of these transformations are described in detail by Bacon et al. in [BGS94]. They are omitted here, except for two highly related transformations. These transformations are software pipelining and prefetching and are briefly described further.

Software pipelining, as well as loop unrolling, is a technique for improving instruction level parallelism. Similar to hardware pipelining, described in section 2.1.2, software pipelining transforms a loop in such a way that each iteration of the new loop contains instructions from several different iterations of the original loop. This transformation requires a start-up code before the loop to fill up the software pipeline and an additional code after the loop to process the remaining elements of the loop. Software pipelining exploits the ILP across different loop iterations, thus, allowing instructions from successive iterations to execute in parallel. Software pipelining and loop unrolling can both achieve better scheduling for the inner loop, but in a different way: loop unrolling tackles branch and counter update overhead whilst software pipelining attempts to reduce the time when the inner loop is not running at a peak speed. Figure 3.8 presents two graphs from [BGS94] that show how operations are overlapped in the inner loop after loop unrolling and software pipelining. The shaded area in these graphs shows when the loop is not running at the peak speed. For loop unrolling, this happens during each iteration, whilst for software pipelining it happens only at the beginning and the end of the loop. Originally, software pipelining was intended to be used on VLIW platforms as described by Lam in [Lam88]. However, it was later noticed that loops on RISC and

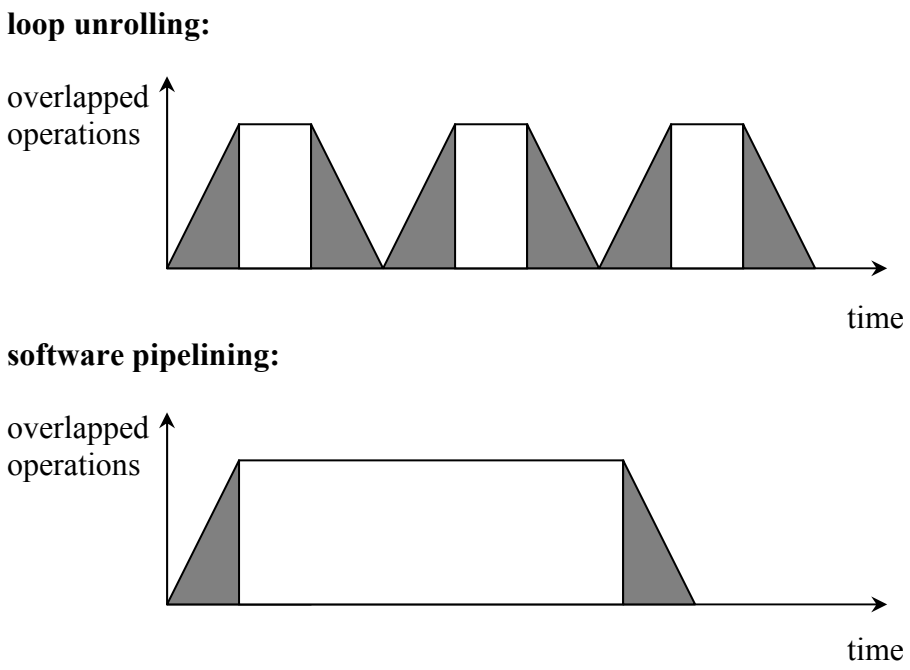


Figure 3.8: Software pipelining versus loop unrolling

other platforms could also benefit from this transformation. Allan et al. generalised and thoroughly analysed this transformation in [AJL+95].

Software prefetching [CKP91] is a technique that can reduce the number of compulsory misses by inserting prefetch instructions into the code to bring data into the cache before it is needed. In this case, when data is accessed it is already in the cache and thus, reduces potential stalls. This technique can considerably improve program performance. However, inserting more instructions can potentially degrade performance. Therefore, analysis is needed to determine whether the code will benefit from the software prefetching. It is important to find the best places where to insert prefetch instructions in code and the amount of data to be brought into cache. Mowry et al. and VanderWiel et al. describe and analyse various techniques to apply software prefetching in [MLG92] and [VL00] respectively. Figure 3.9 presents a simple software pipelining example from the last paper. A compulsory cache miss occurs in the original loop at every fourth iteration, assuming a four-word cache block. Therefore, a basic approach to prefetching is to fetch data from the main memory to the cache one iteration before this data is needed.

The following section will describe static analysis and code optimisations that use loop tiling, array padding and loop unrolling.

original loop:

```
for (i = 0; i < N; i++)
    ip = ip + a[i]*b[i];
```

loop after software prefetching

```
for (i = 0; i < N; i++){
    fetch( &a[i+1]);
    fetch( &b[i+1]);
    ip = ip + a[i]*b[i];
}
```

Figure 3.9: Software prefetching example for inner product calculation

3.2 Static analysis and optimisations

The previous section described transformations used in the research of this thesis that can improve the program instruction level parallelism or data locality. The major question is how to choose the transformation parameters to achieve the best performance. Therefore, this section surveys work on the static analysis of applications for data locality and ILP. It also reviews static methods for choosing the transformation parameters to improve data reuse and reduce cache misses.

3.2.1 Improving ILP

Program optimisations that improve instruction level parallelism are out of the scope of this research. However, some of them can also improve cache performance as in the case of loop unrolling. It can reduce the number of memory accesses as described in section 3.1.4 and thus, is examined further.

This section briefly reviews existing work on selecting unrolling factors for loops to minimise execution time. Carr and Kennedy describe a technique for automatically choosing the best unrolling factor for a transformation called unroll-and-jam in [CK94]. This transformation consists of two transformations: loop unrolling and loop fusion. First, loop unrolling is applied to an outer loop and then

original loop:

```
      DO 10 I = 1, 2*M
        DO 10 J = 1, N
10        A(I) = A(I) + B(J)
```

after unroll-and-jam of I by a factor of 1:

```
      DO 10 I = 1, 2*M, 2
        DO 10 J = 1, N
          A(I) = A(I) + B(J)
10        A(I+1) = A(I+1) + B(J)
```

Figure 3.10: Unroll-and-jam transformation example

loop fusion is applied to bring inner loops together, as shown in the example from this paper in figure 3.10.

The authors propose an algorithm for automatically transforming a loop to improve performance by optimising the ratio of memory operations to floating-point operations. It is based on a static method of estimating the performance of a loop on a targeted platform using a simple performance model that incorporates only a few parameters of that platform, such as the number of floating-point and load operations per cycle and the number of registers. The general idea of this method is to optimise a loop in such a way that both memory accesses and floating-point operations are performed at peak speed without delays.

Two characteristics are used to analyse and quantify the balance between loads and floating-point operations: machine balance and loop balance. Machine balance is a platform-dependent characteristic, defined as the following:

$$\text{Machine balance } (=\beta_M) = \frac{\text{max words / cycle } (=M_M)}{\text{max flops / cycle } (=F_M)},$$

where M_M is the peak rate of data loads and F_M is the peak rate of floating-point operations. Loop balance is a characteristic of a specific loop, defined as the following:

$$\text{Loop balance } (\beta_L) = \frac{\text{number of memory references } (=M)}{\text{number of flops } (=F)}.$$

Comparing loop balance with machine balance allows one to analyse the performance of a particular loop on a particular platform. The case when $\beta_L < \beta_M$

means that the loop is compute bound, i.e. the rate of data retrieval from memory is faster than its processing rate. The case when $\beta_L > \beta_M$ means that it is memory bound, i.e. data is retrieved from the memory slower than it can be processed. Finally, the loop is balanced on the target platform if $\beta_L = \beta_M$. Therefore, to improve performance with unroll-and-jam transformation, a non-linear integer optimisation problem should be solved that improves the balance of the loop:

objective function: $\min \left|\beta_L - \beta_M\right|_0$

constraint: # floating-point register required $\leq$ register-set size

The last constraint is needed as the unroll-and-jam transformation can potentially spill floating-point registers. Loop balances and register usage are calculated at compile time as functions of X_i, that is the number of times the i^{th} outermost loop is unrolled + 1. Hence, to determine the best possible unrolling factor, the above optimisation problem is solved to get a linear function of X_i and then the solution space is searched in parallel with checking the register pressure.

The above method has been implemented in a Fortran source-to-source compiler. The experimental results showed that in most of the cases hand optimisation was unable to achieve a much better balance than this automatic technique. However, the loop performance prediction model used in this method is very simple, based only on a few parameters of a target platform, such as the number of registers and machine balance. Hence, it may not predict performance correctly on current platforms, with processors supporting out-of-order execution and with memory hierarchy. To overcome this problem Carr and Guan propose an extension to this algorithm in [CG97]. In the new method, the calculation of β_M is the same, but the calculation of β_L reflects the observation that cache miss latency can be hidden in some platforms by software prefetching or non-blocking caches. The architecture in this paper is assumed to have a prefetch-issue buffer of the size $P_M \geq 0$ and a prefetch latency of $L_M \geq 0$ cycles. Therefore, the prefetch issue bandwidth is $I_M = \frac{P_M}{L_M}$. Considering, that for every L_L cycles the innermost loop requires P_L prefetches, then in the case of $I_L = \frac{P_L}{L_L} \leq I_M$, the memory latency can be hidden, otherwise all prefetches

($P_L - I_M L_L$) cannot be processed. This gives the final version of calculating the loop balance:

$$\beta_L = \frac{M_L + (P_L - I_M L_L)^+ \times \frac{C_m}{C_h}}{F_L},$$

where $x^+ = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$, C_m is the cost of a cache miss and C_h is the cost of a cache hit. In addition, to compute the memory operation cost, data reuse is analysed. This is described in the next section.

The modified method takes more parameters of the targeted platform into consideration and is potentially more precise than the previous one. Finally, Sarkar presents a similar algorithm in [Sar00], which uses a cost function that incorporates unrolling factors for all loops. This algorithm automatically determines the best unrolling factors for perfectly nested loops and generates more compact code than the algorithm described in [CG97]. It enumerates a set of all profitable unroll vectors during the optimisation process and computes the cost function for each of them. Finally, the unrolling vector with the smallest cost function is selected.

3.2.2 Data locality analysis and optimisations

Data locality is an important characteristic of the program that describes the program's ability to effectively utilise the memory hierarchy. Analysing data locality statically allows one to predict the memory behaviour and to find the potential for utilising the cache hierarchy. It can be further used to apply transformations that improve data locality and to speed up the program, as is shown in the next subsections.

Most of the research in this domain is aimed at analysing data reuse that occurs within loops. Wolf and Lam describe a mathematical formulation of reuse and locality in [WL91a], based on the loop transformation theory that is briefly described in paragraph 3.1.1 of this chapter. They propose an algorithm for improving the data locality of a loop nest.

The authors of this paper stress a distinction between reuse and locality: if some data is used in different iterations of a loop nest, it is reused in this loop nest.

However, reuse does not guarantee locality, since the data may be flushed out of the cache by intervening iterations. Those iterations that can exploit reuse form a localised iteration space. This iteration space can be characterised as localised vector space to abstract from its bounds.

This paper describes four types of reuse: self-temporal reuse, when a reference accesses the same word for different loop iterations; self-spatial reuse, when a reference accesses a word in the same cache block for different loop iterations; group-temporal reuse, when references accesses the same word and group-spatial reuse, when references refer to a word in the same cache block. The following loop nest is an example from this paper to demonstrate these types of reuse:

for I_1 := 1 **to** n **do**
  for I_2 :=1 **to** n **do**
    f(A[I_1],A[I_2]);

Reference A[I_1] from this loop nest has self-temporal reuse in the innermost loop and reference A[I_2] has self-temporal reuse in the outermost loop. Besides, reference A[I_1] has self-spatial reuse in the outermost loop and reference A[I_2] has self-spatial reuse in the innermost loop.

This paper uses the concept of uniformly generated references to quantify reuse and locality, and underlines that non-uniformly generated references exhibit little exploitable reuse. The definition of the uniformly generated references is the following:

Let n be the depth of a loop nest, and d be the dimensions of an array A. Two references $A[\vec{f}(\vec{i})]$ and $A[\vec{g}(\vec{i})]$, where $\vec{f}$ and $\vec{g}$ are indexing functions $Z^n \rightarrow Z^d$, are called uniformly generated if

$$\vec{f}(\vec{i}) = H\vec{i} + \vec{c}_f \text{ and } \vec{g}(\vec{i}) = H\vec{i} + \vec{c}_g$$

where H *is a linear transformation and* $\vec{c}_f$ *and* $\vec{c}_g$ *are constant vectors.*

The references to the same array and with the same H are further partitioned into equivalence classes of references called uniformly generated sets. Consider the following sample loop nest from this paper:

for I_1 := 0 **to** 5 **do**

for I_2 :=0 **to** 6 **do**

A[I_2+1] := 1/3 * (A[I_2] + A[I_2+1] + A[I_2+2]);

References A[I_2], A[I_2+1] and A[I_2+2] have the following indexing functions:

$$\begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}I_1\\I_2\end{bmatrix}+\begin{bmatrix}0\end{bmatrix},\ \begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}I_1\\I_2\end{bmatrix}+\begin{bmatrix}1\end{bmatrix} \text{ and } \begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}I_1\\I_2\end{bmatrix}+\begin{bmatrix}2\end{bmatrix}.$$

These references have the same H = $\begin{bmatrix}0 & 1\end{bmatrix}$ and therefore belong to the same uniformly generated set.

This paper further presents methods for quantifying four types of reuse within a loop nest: self-temporal, self-spatial, group-temporal and group-spatial. For example, a reference $A[H\vec{i}+\vec{c}]$ has a self-temporal reuse if iterations $\vec{i}_1$ and $\vec{i}_2$ access the same data, that is $H\vec{i}_1+\vec{c}=H\vec{i}_2+\vec{c}$, or $H(\vec{i}_1-\vec{i}_2)=\vec{0}$. In this case, the reuse occurs in the direction of vector $\vec{r}$, if $H\vec{r}=\vec{0}$. The solution of this equation is a vector space $R_{ST}=\ker H$, called self-temporal reuse vector space. The condition for the reuse to be exploited is the inclusion of direction vectors in the localised vector space. Other types of reuse are quantified in a similar way.

Finally, this paper presents methods for calculating the number of memory accesses per iteration for the innermost loop and introduces an algorithm for improving locality. This algorithm uses loop interchange, reversal, skewing and tiling to improve cache performance. It attempts to place outermost loops without reuse and then tries to tile innermost loops to minimise the memory accesses per iteration. This algorithm is evaluated using LU-decomposition, matrix-multiplication and SOR benchmarks.

The method described above provides a practical solution for quantifying the data locality of loop nests and for improving cache performance. However, it can use only unimodular transformations and can be applied only to perfectly nested loops. Affine partitioning is used to overcome this restriction, as described by Lim et al. in [LLL01]. It allows transforming arbitrary loop nests and improving their data locality. This paper generalises loop tiling and extends the data locality algorithm presented in [WL91a]. The locality optimisations have been evaluated on a number of kernels and speed-up has been achieved in each case.

Bodin et al. use the concept of "reference window" to optimise program data locality in [BJW+92]. Reference window characterises "active" array elements that have a reuse and therefore has to be kept in the cache. It is defined as following:

> *The* **reference window**, *W(t), for a dependence between two references to array A,* $\Delta_A : S_1 \rightarrow S_2$, *at time t is defined to be the set of all elements of A that are referenced by* S_1 *before t that are also referenced after or at t by* S_2.

Further, a cost and a benefit of a reference window are defined:

> *The* **cost** *of a reference window Cost(W) is defined as the maximum size of the window over the time (the size of the window W is denoted* $\|W\|$*).*
>
> *The* **benefit** *of a reference windowBen(W) is defined as the number of accesses to main memory saved.*

Consider the following loop from this paper, for example:

```
      DO 1 i1 = 1, N1
S1        A(i1) = X(i1)
S2        D(i1) = X(i1-3)
1     CONTINUE
```

This loop has the reference window $W_X = \{X(i_1\text{-}3), X(i_1\text{-}2), X(i_1\text{-}1)\}$. Its $Cost(W_X) = 3$ and $Ben(W_X) = N_1\text{-}3$. If the reference window fits the cache, the data locality of a loop nest is optimal. Otherwise, the window has to be reduced using loop nest restructuring to fit it into lower levels of the memory hierarchy.

The size of the reference windows is further used to drive data locality optimisations. Loop interchange and loop tiling are used to reduce the size of the reference windows to fit into the cache. The practical optimisation algorithm presented in this paper is evaluated on a number of hand-coded benchmarks and speed-up is achieved in most of the cases. The major restriction of this algorithm however, is that it can be applied only to perfect loop nests and it uses approximations to express reference windows analytically in order to simplify calculation.

McKinley et al. use a simplified cost model for computing temporal and spatial reuse for loops and to improve data locality in [MCT96]. First, references within a loop are placed into the specific reference groups. Reference groups consist of those

INPUT:

L = { $l_1, \ldots, l_n$ } a loop nest with headers $lb_l, ub_l, step_l$

R = { $Ref_1, \ldots, Ref_m$ } representatives from each reference group

$trip_l$ = ($ub_l - lb_l + step_l$) / $step_l$

cls = the cache line size in date items

$coeff(f, i_l)$ = the coefficient of the index variable i_l in the subscript f

$stride(f_1, i_l, l)$ = $\left| \; step_l * coeff(f_1, i_l) \; \right|$

OUTPUT:

$LoopCost(l)$ = number of cache lines accessed with l as innermost loop

ALGORITHM:

$$\textbf{LoopCost}(l) = \sum_{k=1}^{m} \textbf{RefCost}(Ref_k(f_1(i_1, \ldots, i_n), \ldots, f_j(i_1, \ldots, i_n)), l) \prod_{h \neq l} trip_h$$

$$\textbf{RefCost}(Ref_k, l) = \begin{cases} 1 & \text{if } ((coeff(f_1, i_l) = 0) \wedge \ldots \wedge (coeff(f_j, i_l) = 0)) \quad \textbf{Invariant} \\ \dfrac{trip_l}{\left(\dfrac{cls}{stride(f_1, i_{l,l})}\right)} & \text{if } ((stride(f_1, i_l, l) < cls) \wedge (coeff(f_2, i_l) = 0) \wedge \ldots \wedge (coeff(f_j, i_l) = 0)) \quad \textbf{Unit} \\ trip_l & \text{otherwise} \quad \textbf{None} \end{cases}$$

Figure 3.11: Algorithm for calculating loop cost (McKinley et al.)

references that have group-temporal or group-spatial reuse. Then, the cost of the reference groups and loops is calculated, as shown in figure 3.11.

First, *RefCost* calculates the number of cache lines used by the loop l for the reference Ref_k. For loop-invariant references, *RefCost* is equal to 1. For consecutive references, RefCost is equal to *trip/(cls/stride)*, where *trip* is the number of iterations in the loop, *cls* is the size of the cache line and *stride* is the loop step multiplied by the coefficient of the loop index variable. For non-consecutive references, *RefCost* is equal to *trip*. Finally, *LoopCost* calculates the total number of cache lines accessed within the loop nest, when l loop is the innermost position. This function can be used to guide loop nest transformation by interchanging loops in order to minimise the number of accessed cache lines. Figure 3.12 presents an example from the paper for calculating *LoopCost* for matrix multiplication kernel. Arrays C(I,J), A(I,K) and

```
{ JKI ordering }
DO J = 1, N
    DO K = 1, N
        DO I = 1, N
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

LoopCost (with *cls*=4)

Refs	J	K	I
C(I,J)	$n * n^2$	$1 * n^2$	$\frac{1}{4} n * n^2$
A(I,K)	$1 * n^2$	$n * n^2$	$\frac{1}{4} n * n^2$
B(K,J)	$n * n^2$	$\frac{1}{4} n * n^2$	$1 * n^2$
Total	$2n^3 + n^2$	$\frac{5}{4} n^3 + n^2$	$\frac{1}{2} n^3 + n^2$

Figure 3.12: Loop cost for matrix multiplication kernel (McKinley et al.)

B(K,J) are invariant for the loops K, J and I respectively and therefore have *RefCost=1*. The same arrays have n non-consecutive references for the loops J, K and J again respectively and therefore have *RefCost=n*. Finally, these arrays have n consecutive references for the loops I, I and J respectively and therefore have *RefCost*=$\frac{1}{4}n$. The *LoopCost* function shows that this loop nest accesses minimum number of cache lines when the loop *I* is the innermost.

Furthermore, an algorithm for improving data locality by combining loop permutation, fusion, distribution and reversal is presented in this paper. It is a relatively simple and inexpensive algorithm for minimising the cost function. It can be applied to non-perfectly nested loops with complex subscript expressions. This algorithm has been evaluated on a wide range of programs. It achieved a significant performance improvement on several of these programs.

Finally, Ghosh et al. present an algorithm for calculating the cache misses precisely for a loop nest using Cache Miss Equations in [GMM98]. Initially, a cache set accessed by a reference R_A at iteration $\vec{i}$ is calculated as following:

$$Mem_{RA}(\vec{i}) = Memory\_Address\_of\_R_A(\vec{i})$$

$$Memory\_Line_{RA}(\vec{i}) = \lfloor Mem_{RA}(\vec{i}) / L_s \rfloor$$

$$Cache\_Set_{RA}(\vec{i}) = \lfloor Mem_{RA}(\vec{i}) / L_s \rfloor \bmod N_s,$$

where L_s is the cache line size, N_s is the number of cache sets and $Mem_{RA}(\vec{i})$ is the memory address accessed by R_A at the iteration $\vec{i}$. $Mem_{RA}(\vec{i})$ can be computed by analysing the subscript expressions of R_A. Consider the matrix multiplication example shown in figure 3.12. If the number of cache sets is 128, line size is 4, the base address of the array C is 4192 and the number of elements per column of this array is 32, then the cache set accessed by the reference C(j, i) is the following:

$$\lfloor (4192 + 32i + j - 1) / 4 \rfloor \bmod 128$$

Then reuse is analysed for a loop nest and reuse vectors are generated in a similar way to [WL91a], which is briefly reviewed above. Furthermore, misses are quantified along the reuse vector and two types of CME equations are generated for a particular reuse vector of a particular reference: cold miss equations and replacement miss equations. Solutions to these equations represent a potential number of compulsory and conflict misses respectfully. Finally, an algorithm is presented for quantifying all the cache misses of a loop nest by combining multiple CMEs.

Cache Miss Equations allow a precise analysis of cache misses and use mathematical analysis to determine solutions. This paper presents algorithms to find optimal optimisations by solving CMEs before and after applying particular transformation. However, solving all equations can be a potentially time consuming process, slower than other approximate static methods described above. Vera and Xue extend CME framework in [VX02] to analyse the cache behaviour of whole programs that have regular computations and validates the accuracy of the method on a number of real codes from SPEC'95 benchmark in comparison with cache simulation for those codes. It also shows that for large programs, such as applu, the whole analysis on Pentium III 933MHz takes about 128 seconds, which is about three orders of magnitude faster than the cache simulator, but still slower than other static methods.

3.2.3 Reducing conflict misses

Conflict misses occur due to the limited cache associativity, as described in chapter 2. It is possible to considerably reduce conflict misses by increasing the associativity of the cache on the hardware level. However, it is an expensive solution. It is also possible to use software optimisations to considerably reduce conflict misses, particularly by using array padding and loop tiling.

Temam et al. present a comprehensive analysis of cache interferences in numerical loop nests in [TFJ94], which detect and compute the number of conflict misses analytically. The method is based on introducing reuse and interference sets and on counting the number of cache misses, when a disruption of locality occurs. The paper shows that the algorithm is fast and reasonably precise by evaluating it on a number of kernels. It also demonstrates that optimising codes for capacity misses only may not be enough, as the conflict misses may be large and frequent.

Lam et al. analyse the influence of loop tiling (blocking) on cache performance in [LRW91]. The authors describe methods for modelling cache interference and provide algorithms for determining the overall cache miss rate as a combination of three types of misses: intrinsic misses, self-interference misses (conflicts between elements of the same array) and cross-interference misses (conflicts between different variables). The input parameters for this algorithm are the matrix size N and the cache size C. The output is the largest tile size that removes self-interference misses. This algorithm is based on finding those array elements that are mapped to the same location in the direct-mapped cache. For an array word Y[i,j] it attempts to find another array word of the form Y[$i + di$,$j + dj$] that maps to the same location in the cache of the size C. Finally, the returned best tile is the maximum of di and dj. It is a relatively fast and simple method, which is easy to implement. However, it is also imprecise and does not remove conflict misses that may occur between different arrays. An additional approach to the above algorithm, for eliminating cache misses by copying reusable non-contiguous data into contiguous area, is presented in this paper and is called copy optimisation. For example, array tiles can be copied to some temporary continuous arrays that do not exhibit cache conflicts.

Copying all array tiles into temporary arrays can also degrade performance when the copying overhead is higher than the benefit from reducing conflict cache misses.

Therefore, Temam et al. extends this work and presents a compile-time technique in [TGJ93] for selective data copying by analysing the cost and the benefit of eliminating conflicts. Cross interferences are further categorised as internal and external ones. Finally, an algorithm targeting each particular type of interference step by step is presented and is manually evaluated on a number of benchmarks. However, this algorithm may not be precise on modern processors where the cache miss latency can be hidden by the execution of other instructions.

Coleman and McKinley present a Tile Size Selection (TSS) algorithm in [CM95], based on the cache size and cache line size, to eliminate both capacity and conflict misses. This algorithm uses rectangular tiles and attempts to determine the best dimensions of these tiles without self-interference. First, potential row sizes to fit in the cache are determined and potential column sizes are determined as multiples of the cache line size to benefit from spatial locality. Finally, those dimensions for the tile are chosen that minimise the number of cross-interference. This algorithm is evaluated on a number of kernels and its accuracy is validated using simulators. The rate of conflict misses is reduced considerably in most of cases.

Rivera and Tseng improve the above algorithms and integrate intra-variable padding in [RT99]. The algorithm for determining tile sizes is based on the one proposed in [CM95], however, it is simpler and more accurate. Calculation of both the height and width of the tile uses a recursive function and can be computed simultaneously. In some pathological cases, frequent conflict misses can still occur between tiles. In such cases, intra-variable array padding can solve the problem. It is incorporated into a cost model with loop tiling so that for each padding parameter from a small range, tile sizes are calculated and the ones that minimise conflict misses are chosen. This algorithm is evaluated on matrix multiplication and LU-decomposition for various matrix sizes and performance improvement is achieved in most of the cases.

Finally, the paper [GMM98], briefly reviewed in the previous section, demonstrates how to use Cache Miss Equations to automatically determine intra-variable and inter-variable padding to reduce self- and cross-interferences. An algorithm to determine optimal tiling parameters, which reduce conflict misses by

combining array padding with loop tiling, is further proposed. Its strong point is that it provides precise information about cache misses. However, it is considerably slower than all the above algorithms.

3.2.4 Reducing compulsory misses

Two previous sections concentrated on algorithms to remove capacity and conflict misses. This section will only briefly review algorithms that use software prefetching to reduce compulsory misses, as it is out of the main scope of the research of this thesis.

The major challenge for designing algorithms for software prefetching is to identify data that has to be prefetched and to determine when this data should be prefetched. The potential problems are the software prefetching overhead and cache disruption if prefetching instructions are not scheduled correctly.

Callahan et al. present a theoretical algorithm in [CKP91] to identify data that should be prefetched, based only on the analysis of variables within the inner loops. Its influence on performance and hit ratio is evaluated using a simulator. Methods for reducing the overhead and for eliminating unnecessary prefetches are proposed. They are based on the analysis of the dependence graph in an attempt to eliminate prefetching data that already resides in the cache.

Mowry et al. describes a practical compiler algorithm for prefetching in [MLG92]. It uses a similar framework for data locality analysis as in [WL91a] and reviewed in section 3.2.2. This analysis allows one to determine accesses that may cause cache misses to be candidates for prefetching. A loop splitting transformation is used further to split the innermost loop into a prolog loop, steady state loop and epilog loop. The first loop initialises the cache; the steady state loop executes the original loop iterations and prefetches data for the further iterations; the epilog loop finalises the execution of the last iterations. Software pipelining transformation, briefly described in section 3.1.5, is applied to the split loops to ensure that there are enough iterations before prefetched data is used. The algorithm has been evaluated on a number of benchmarks and performance improvements have been achieved in most of the cases. However, the restriction of this algorithm is that it can handle only affine array accesses. It was also noted from experiments that conflict misses

exhibited in some programs could considerably suppress the benefit from software prefetching.

Finally, VanderWiel and Lilja survey and compare various data prefetch mechanisms and describe their drawbacks and benefits in [VL00].

3.3 Dynamic analysis

The two previous sections presented techniques to analyse a program cache performance statically and described transformations that could improve the cache behaviour of programs. Some of the techniques presented proved useful for a variety of codes. However, most of these techniques are inherently imprecise as they use simplified program models in order to be reasonably fast and tractable. Therefore, they are usually restricted to specific types of loops and memory access patterns; otherwise, they can be time consuming as in the case of CMEs. This section presents dynamic techniques that attempt to overcome some of these problems, and are used to analyse the program performance during execution or during simulation. The major benefit of these methods is the access to run-time information, which is not available at compile time. Dynamic analysis has its own advantages and restrictions, and is not intended to replace static analysis. Instead, both static and dynamic analysis can complement each other, as is shown in this and the following section 3.4.

3.3.1 Profiling

Profiling is a wide-spread technique for obtaining various run-time program parameters during its execution. One of the simplest and basic techniques is a procedure-level profiling when code is instrumented by adding calls to the monitoring routines on the entry and the exit of each profiled procedure. After the code is executed, the profiling information is gathered into the file that may be further parsed to obtain the program execution time distribution. Such techniques are easy to implement but overhead due to calls to the monitoring routines can be excessively high. Besides, time measuring can be more complicated on time-sharing platforms, since execution time of other processes needs to be accounted. Therefore,

sampling technique is often used on such systems. This technique samples the value of the program counter with some interval and later obtains execution time statistically from the distribution of the samples within the whole program. It is used in such tools as gprof [GKM82], has a relatively small overhead and is useful in determining parts of the program that dominate the execution time, thus, reducing unnecessary analysis of the whole program. Unfortunately, procedure-level profiling is insufficient to spot the problems inside the subroutines. This can be important in cases when those subroutines contain multiple loop nests or irregular memory accesses, which are difficult to analyse statically. Therefore, other tools are used that can profile programs on a basic block level or on an instruction level.

Smith describes a tool called Pixie in [Smi91] that allows the collection of run-time information about basic blocks of the program. This tool rewrites the executable file and inserts additional instructions to count the number of executions of each basic block. During the execution of the modified code, the run-time information is captured and saved into data files. After the execution is finished, the tool analyses the data files and produces a report about cycle counts within subroutines. This information can be matched with the source code and can be used to determine the bottlenecks within the procedure that should be further optimised. The major problem of this method arises in profiling programs with a large number of small basic blocks. In such cases, the number of inserted instructions can be overwhelming, and, for example, can influence the behaviour of caches, thus, producing imprecise cycle counts. Besides, this tool cannot provide information about stalls, which could be useful for program optimisations. To overcome this problem, new methods are used for profiling programs on an instruction level without their modification. These methods are based on using processor hardware counters. These counters obtain run-time information in parallel with the program execution and can dump this information periodically to the collecting tool to be saved for further analysis. Such methods do not influence the behaviour of the program and thus, can produce accurate instruction-level profiles with a small overhead.

Anderson et al. describe the Digital Continuous Profiling Infrastructure (DCPI) system that works on Alpha processors and uses their hardware performance counters in [ABD+97]. It consists of two parts: a data collection subsystem and an

analysis subsystem. The data collection subsystem runs continuously on a platform and collects profiles for unmodified executables or even for the entire system. It samples performance counters for various events, such as cache misses or branch mispredictions, periodically at a high rate (over 5200 samples per second on a 333MHz Alpha processor) with a low overhead (1-3% slowdown) and records them in a database. The analysis subsystem produces accurate information about the program based on the collected profile data at several levels: from the time spent in subroutines to the number of stalls for each instruction within the subroutine. Furthermore, it can provide an explanation for particular stalls in the program. However, the major restriction of the DCPI tool is that it fails in attributing profile data to the instructions on out-of-order execution processors.

Dean et al. present a tool called ProfileMe in [DHW+97] for instruction-level profiling on out-of-order execution processors. Unlike DCPI that counts processor events, ProfileMe samples instructions and collects information about stalls and events within a pipeline. It can also collect information about parallel execution and interaction of concurrent instructions. Besides, ProfileMe can provide information not only about instructions retired from the pipeline, but also about instructions that have been aborted due to speculative execution, thus, providing valuable information for further optimisations. A special inexpensive hardware support is needed for such profiling and is available in the latest Alpha processors. This is done by adding a few ProfileMe registers for recording the processor state for a profiled instruction and by passing a special ProfileMe tag through a pipeline to indicate profiling instructions. Therefore, this tool enables accurate low-overhead instruction-level profiling on both in-order and modern out-of-order execution platforms to provide feedback about stalls in the pipeline and about useful concurrency and is useful to drive further optimisations. Another tool, called VTune [Int03b], provides similar instruction-level profile information on Pentium-based platforms.

The tools described above provide raw information about stalls in programs and are able to identify bottlenecks in these programs. However, they do not provide information such as the number of cache misses for a particular instruction, most notably in the case of dynamically allocated memory, and the types of cache misses, which could be useful for further memory performance optimisations. Therefore,

additional methods are needed to analyse the nature of cache misses using obtained profile information.

Buck and Hollingsworth describe a technique for determining the number of cache misses for a particular memory area that can be allocated statically or dynamically in [BH00]. It uses hardware counters that can cause interrupts after a number of cache misses and can report the address of the last cache miss. This address is associated with the memory region defined by the program and the corresponding counter is incremented. Therefore, this technique provides information about program objects that have poor cache behaviour. The SPLAT tool, described by Sánchez and González in [SG00], is able to identify the type of cache misses by matching static information about program data locality provided by the compiler with the run-time information provided by a profiler. It uses fast methods for analysing data locality, described in section 3.2 of this chapter, and fast profiling techniques, thus, providing information about cache misses with a low overhead. Such information can be used further in choosing particular types of optimisation.

Finally, a tool called ATOM described by Srivastava and Eustace in [SE94] should be noted. It is used for instrumenting programs on an Alpha platform to obtain various precise run-time parameters and can be used for building customised program analysis tools. It has its own macro language to define procedures, basic blocks and instructions and can insert calls to auxiliary subroutines with register value parameters on instruction or basic-block level, for example. The information obtained can be stored in temporary arrays and can be dumped onto disk after the execution of the program. This information can be further used by various analysis tools. The ATOM tool is particularly useful in designing simulators, as described in the following section.

3.3.2 Simulating

System simulators can assist in understanding the run-time behaviour of the program and the influence of various architectural parameters on the program execution. They are useful in cases where static information is unavailable and profiled information is imprecise or not sufficient for successful optimisation. Such tools contain a software model of the hardware and simulate the execution of the

program step by step. The major advantages of the simulation are that it provides the opportunity to analyse and visualise the hardware state step by step during the program execution; to test new hardware designs and validate performance and to obtain various run-time parameters. However, its major drawbacks are high execution time and resource consumption, and the need to have a precise system model.

Burger et al. and Austin et al. describe a tool called SimpleScalar for computer system modelling in [BAB96] and [ALE02]. This tool contains flexible software models for different hardware to help designers test their ideas before building the real system. These models include a dynamic program analyser, a branch predictor simulator, a multilevel cache memory simulator and many others, and are characterised by performance, flexibility and detail. The general trade-off for the models is that the higher the detail level and flexibility, the lower the performance. The SimpleScalar tool supports multiple platforms and has a visualisation module capable of displaying the processor pipeline stages for each instruction. This tool is useful in detecting and analysing various software and hardware bottlenecks. The high level of detail and accuracy makes it possible to simulate the behaviour of complex out-of-order execution superscalar processors and cache memories to analyse the cache hit rate, memory access latency or even calculate power dissipation, for example.

Trace-driven simulation is another approach that creates streams of instrumented instructions, which are further used in hardware or software timing models. During the execution of those instruction streams, traces with various parameters are collected. Since these traces can be large, a trace reduction mechanism is used to make them smaller. Finally, the obtained traces are processed to derive useful information. Uhlig and Mudge survey and compare, in detail, over 50 trace-driven simulation tools with the emphasis on memory design in [UM97]. These tools record sequences of memory references and attempt to predict memory-system performance. They are characterised by the detail and accuracy of simulation and how traces are collected and reduced.

Both execution- and trace-driven simulations are generally accurate. However, they require excessive simulation times and computer resources. For example, the

simulators reviewed in [UM97] had a slowdown in the range of 45 to 6250 times compared to the execution time of the original program. Another statistical approach for modelling the performance of superscalar processors, that is both fast and reasonably accurate is presented by Noonburg and Shen in [NS97]. The main idea of this method is to calculate the probability of being in a particular processor state. The processor model consists of blocks or components that have one input and one output, and are interconnected between each other. Therefore, each component has and input and output instruction flow. This flow can be limited by various restrictions, such as the bandwidth of the connection or by being blocked by some components. The processor state is represented as a vector that describes instructions in each component. Finally, a state distribution is computed using Markov chains. This paper presents several simple processor models and compares the performance obtained, using the above method, with the simulated performance. The results demonstrate that the statistical approach can be reasonably accurate (within 2% in 3 benchmarks and within 10% for Livermore loops) and is considerably faster than execution- or trace-driven simulations.

To complete this section about simulation techniques, some examples of their usage are further presented. McKinley and Temam analyse and quantify the loop nest locality of different benchmarks in [MT96] by simulating the cache and by using the ATOM tool to obtain information about data accesses. The intra- and inter-nest locality are measured and quantified in a similar way as described in section 3.2.2. Though the simulation process was excessively slow, it made it possible to obtain precise information about the number of cache misses and their types for loop nests. The analysis of the obtained data questioned some common assertions such as “spatial reuse is the dominant form of reuse” and that “capacity misses occur more frequently than conflict misses, and both are significant sources of misses”. For example, it was found that spatial and temporal reuse are generally balanced and that group-conflict misses dominate intra-nest misses. Besides, this paper confirmed another common assertion that most reuse occurs within a nest rather than across nests. However, it also shows that most of the misses occur across nests. Such results can help in designing or modifying cache systems and can be useful for program optimisations.

Lebeck and Wood describe a cache profiling tool, called CPROF in [LW94]. This tool is a uniprocessor cache simulator and visualiser that allows one to detect the number and type of cache misses on the source-line level. It can further suggest program transformation such as padding, loop fusion, blocking and others to improve performance. This tool is evaluated on a number of SPEC benchmarks and performance improvement is achieved in most of the cases.

Van der Deijl et al. present a Cache Visualisation Tool (CVT) in [VKT+97] that visualises the cache operations step by step. This tool uses a cache simulator to produce detailed analysis of the cache behaviour for various code structures and can be used to analyse the influence of different program transformation on this behaviour. Besides, it is possible to study the effect of cache designs on program performance by changing cache parameters of the simulator. Therefore, this tool can be useful for hardware and software optimisations. Finally, Yu et al. describe a technique for visualising the cache behaviour and reuse distances for the whole program as a compact pattern in [YBH01]. This information can be further used for global program optimisations.

3.4 Dynamic optimisations

The previous section presented various dynamic methods to analyse the run-time behaviour of the program. This section describes techniques that use the obtained run-time information to tune the code for better performance during feedback-assisted compilation, or to optimise the program on the fly as during adaptive compilation. It completes the review of the major analysis and optimisation methods related to the research presented in this thesis.

3.4.1 Feedback-assisted and iterative compilation

When run-time information for a program is obtained, it is possible to better optimise this program using information that was unavailable during the compilation stage. An optimisation process that instruments and executes a program for the particular dataset to obtain run-time information and then uses this information to

automatically re-optimise the program is called feedback-assisted compilation. This process is also known as feedback-directed or profile-guided optimisation.

Chang et al. describe a 2-step compiler system in [CMH91] that automatically profiles a program and then optimises this program using profile information. The profiler can identify frequently used program paths and can obtain run-time information about branches taken, loop bounds, etc. This information can further help to improve program performance. For example, it can be used to improve the accuracy of branch prediction and therefore improve performance on modern pipelined superscalar processors that have high penalty for branch misprediction. It can also be used to group frequently executed sequences of basic blocks together to improve instruction cache utilisation and reduce the number of branch instructions. Besides, various other optimisations such as loop unrolling, loop invariant code removal and dead code removal, for example, can produce better quality code taking run-time information into consideration. This paper presents algorithms for applying the above optimisations using profile information. These algorithms are evaluated on a range of benchmarks and performance improvement is achieved in all cases.

Currently, most modern compilers include profile-guided optimisations. Cohn and Lowney describe an implementation of the feedback-directed optimisations in the Compaq compilers for an Alpha platform in [CL99]. Profiles are obtained using either pixie or DCPI tools, which are described in section 3.3.1. Then various optimisations such as inlining, loop restructuring transformations, register allocation, code layout, and branch prediction are preformed and speed-ups are achieved for a number of benchmarks. Finally, Smith reviews various techniques and tools for the feedback-directed optimisations in [Smi00], discusses further challenges and suggests some ways to overcome them.

Current feedback-directed compilation techniques generally optimise the program to improve ILP. However, they do not target the memory bottleneck problem since run-time information still may not be sufficient to choose the best program transformation such as loop tiling or array padding. To overcome this problem, it is possible to create several versions of the program with various tiling and padding parameters, execute them, and choose the best one with better performance. Such a process, that investigates sequences of parameters for various

transformations, creates and executes these variants and picks one with the highest speed-up, is called iterative compilation. Kisuki et al. present an iterative compilation technique in [KKO+00] that uses three program transformations: loop tiling, loop unrolling and array padding. These transformations with varied parameters are applied to a program successively until code with the lowest execution time is obtained. This technique is evaluated on several benchmarks and platforms to demonstrate the performance improvements achieved in comparison with static methods. The major benefits of this technique are the ability to tackle memory problem and the possibility to find optimal code. However, the major drawback of this technique is an excessive optimisation time. Nevertheless, in cases when the lifetime of a program with a particular dataset size is much longer than the optimisation time, it is beneficial to use this technique to obtain code with optimal performance.

Whaley and Dongarra describe "Automatically Tuned Linear Algebra Software" (ATLAS) in [WD98]. This software uses static and iterative techniques to tune various numerical subroutines for a better performance during its first installation. Therefore, all the further calls to these subroutines will be forwarded to a particular optimised variant, depending on the dataset size and other parameters. This is an example of the case when the optimisation time for the library is not critical, since the lifetime of this library is much longer.

Finally, two papers that use the iterative compilation approach, though not directly related to the research of this thesis, should be mentioned. Van der Mark et al. uses iterative compilation in [MRB+99] to optimise programs for embedded VLIW processors. This approach is beneficial for embedded applications as the long optimisation time can be reimbursed by the good performance and by the number of systems produced. Of the major constraints of embedded applications is the limited size of the code. Therefore, an iterative search is used to find the best loop unrolling and software pipelining parameters for a trade-off between code speed and size. Nisbet proposes an iterative search for the best parallelisation transformations on distributed memory architectures using genetic algorithm techniques in [Nis98]. Briefly, these techniques work in a similar way as the evolution of living organisms by iteratively searching better solutions to problems. In the context of parallelisation,

genetic algorithm techniques are used to determine program transformation sequences in order to minimise the execution times of various programs.

3.4.2 Adaptive compilation

Adaptive compilation is a technique for optimising the program dynamically during its execution. The advantage of this technique is that it allows the program to adapt for the particular platform and for the particular dataset to achieve best performance without the need for lengthy recompilations and test executions. However, implementing such a technique in practice is a challenging task since the analysis and optimisation of a program should be performed fast and on the fly, and may degrade performance instead of improving it. Besides, the optimisation tool may not have access to the program source code, and therefore it should be able to transform either intermediate representation of the program or the binary code directly.

Voss and Eigenmann present a framework for dynamic program optimisation called "Automated De-coupled Adaptive Program Transformation" (ADAPT) in [VE00]. It decouples dynamic compilation that produces several versions of the code from the dynamic selection of these versions. This allows compilation to be performed in parallel with program execution to minimise overheads. ADAPT instruments the program to obtain run-time values and then uses a translator that optimises parts of the program during its execution depending on the dataset and platform parameters. When new versions of the program parts are available, the dynamic selection mechanism makes a run-time decision about which version to use in order to achieve better performance. ADAPT supports various transformations including loop distribution, loop tiling and loop unrolling. It is evaluated using three SPEC benchmarks and speed-ups are achieved in all cases in comparison with statically optimised programs.

Zhang et al. describe a tool called Morph in [ZWG+97] for automatically profiling and optimising programs in the background on Alpha platforms. It is composed of three major components. A Morph Monitor profiles programs continuously with low overhead and is similar to the DCPI tool reviewed in section 3.3.1. A Morph Editor is a tool that optimises programs using code layout

optimisations based on the profile information. It deals with the intermediate representation of the program if the source code is unavailable, and transforms it into binary executable form. A Morph Manager analyses the profile information, and makes a decision about when to re-optimise the code. Therefore, the Morph system is capable of optimising programs automatically in the background, taking into account various hardware parameters and program usage patterns. It is evaluated on a number of benchmarks and performance improvement is achieved in all cases.

Finally, Kistler and Franz present a comprehensive analysis for continuous program optimisation in [KF03]. They describe a system that continuously profiles programs and can adjust dynamic data layouts for better cache locality or re-schedule instructions for better ILP in the background with the program execution. An algorithm that decides when the code should be optimised, based on the profiled information, is further presented and is followed by the optimisation algorithms. This system is evaluated on a number of benchmarks and speed-ups are achieved in most of the cases. The results are compared with the statically optimised codes and overheads and profitability of this technique are discussed.

3.5 Summary

This chapter reviewed major papers that are related to the area of memory-hierarchy optimisations. Mathematical models for various transformations and data locality analysis were described in detail to provide an important background for the research presented in this thesis. Various static and dynamic techniques that analyse and improve program performance were also reviewed in detail to be compared with new methods developed in this thesis.

Chapter 4

Iterative Compilation

The focus of this thesis is a platform independent optimisation approach based on feedback-directed program restructuring. This chapter presents the case for iterative compilation. It briefly describes the experimental framework used and shows the influence of various transformations on program performance. The results obtained help explain the difficulty of determining the best transformation parameters using current static or dynamic methods. It is followed by a description of the program optimisation space with the given set of available transformations. An algorithm is proposed for searching the optimisation space of large applications to choose the best transformation to minimise the overall execution time. This algorithm is evaluated on a wide range of kernels and real programs from the SPEC benchmark suite and is compared to existing static and dynamic optimisers.

4.1 Introduction

The research presented in this thesis tackles the problem of the ever-increasing gap between the speed of processor and memory. Previous chapters have provided the motivation and the background for this work and described various hardware and software techniques that attempted to overcome this memory problem.

Briefly, hardware solutions are based on the introduction of faster but smaller intermediate layers of memory between the processor and the main memory. These layers of memory called cache memory exploit data locality. However, the original programs may exhibit many cache misses when the program data is not found in the cache and therefore has to be retrieved from slower main memory. This depends on the program structure and the memory hierarchy organisation, and can considerably degrade performance. Therefore, software optimisation methods based on program transformations are used to improve data locality and reduce the number of cache misses. Three major program transformations are used in this thesis: loop tiling, loop unrolling and array padding. These transformations are capable of reducing cache

misses as described in detail in section 3.1. Potentially, these transformations can be applied manually for small and simple programs. However, it requires a good and detailed knowledge of the underlying hardware from a programmer and is a tedious and time-consuming process. Moreover, any small changes in the software or hardware parameters may invalidate the whole optimisation process so it has to be started from scratch again. Therefore, automatic optimisation approaches are desirable for optimising portable codes for particular architectures.

Traditional automatic optimisation approaches are based on comprehensive static program analysis as described in section 3.2. These approaches attempt to analyse program data locality and to predict the number of cache misses, taking into consideration software parameters and hardware models. Modern platforms have complex internal organisations with the support of pipelines, out-of-order execution and cache memory. Therefore, hardware models used by optimisations are simplified in order for the analysis to be tractable. It means that static approaches provide rough performance estimates and often fail to select the best optimisation. Static approaches also fail in cases where information is not available at compile time. Dynamic methods are intended to solve these problems by obtaining various run-time parameters during program execution and then by re-optimising this program using these parameters. Some of these methods are described in section 3.3. However, it is also shown that current dynamic approaches focus on improving ILP by better branch prediction or on improving instruction cache usage by moving frequently accessed parts of the code closer to each other. Thus, these methods also fail to tackle the problem of the growing performance gap between processor and memory.

This chapter presents an iterative feedback-assisted optimisation approach that can overcome the above problems. This approach is based on creating variants of the program with different transformation parameters. All variants of the program are executed and the one with the lowest execution time is picked as the best version. This algorithm is described in detail further in this chapter and is evaluated on two platforms using a number of kernels and large benchmarks. The major advantage of this approach is that it does not need the detailed knowledge of the program and the underlying hardware and is capable of outperforming current static and dynamic

approaches. Unlike some other iterative compilation techniques that are applied to small kernels, it can successfully optimise large applications using a smart phase order. The major drawbacks are the excessive compilation time of iterative methods and the potential sensitivity of the optimisations to dataset sizes and to conditional dependencies on the data values. However, new techniques for reducing the compilation time have been developed and are presented in chapters 5 and 6. Some techniques that allow applying iterative compilation to programs with different datasets are subjects of the future research and briefly proposed in chapter 6 and 7.

4.2 Experimental framework

This section gives an outline of the experimental framework briefly describing the software architecture, platforms and benchmarks used. Some additional technical details about the platforms used can be found in Appendix A.

4.2.1 Software architecture

The validation of platform-independent iterative compilation and performance prediction techniques developed in this research requires conducting a number of experiments on multiple benchmarks and platforms. Therefore, an optimising software suite capable of conducting a large number of various experiments automatically has been developed. This suite is a set of tools designed to analyse program behaviour and optimise its performance. In order to make this toolset easily usable, portable and flexible, a client/server architecture [Sin92] is used as shown in figure 4.1. It consists of autonomous components such as servers, clients and a shared network file system. These components communicate with each other over the network using data files on the shared file system and using standard telnet protocol [DHP+77] as shown by arrows in figure 4.1. Servers perform various analysis and optimisation tasks on the target platform, while the client is a Graphical User Interface (GUI) application that enables users to interact with servers remotely. A client is platform independent and can access various platforms, obtain and present results in a convenient way.

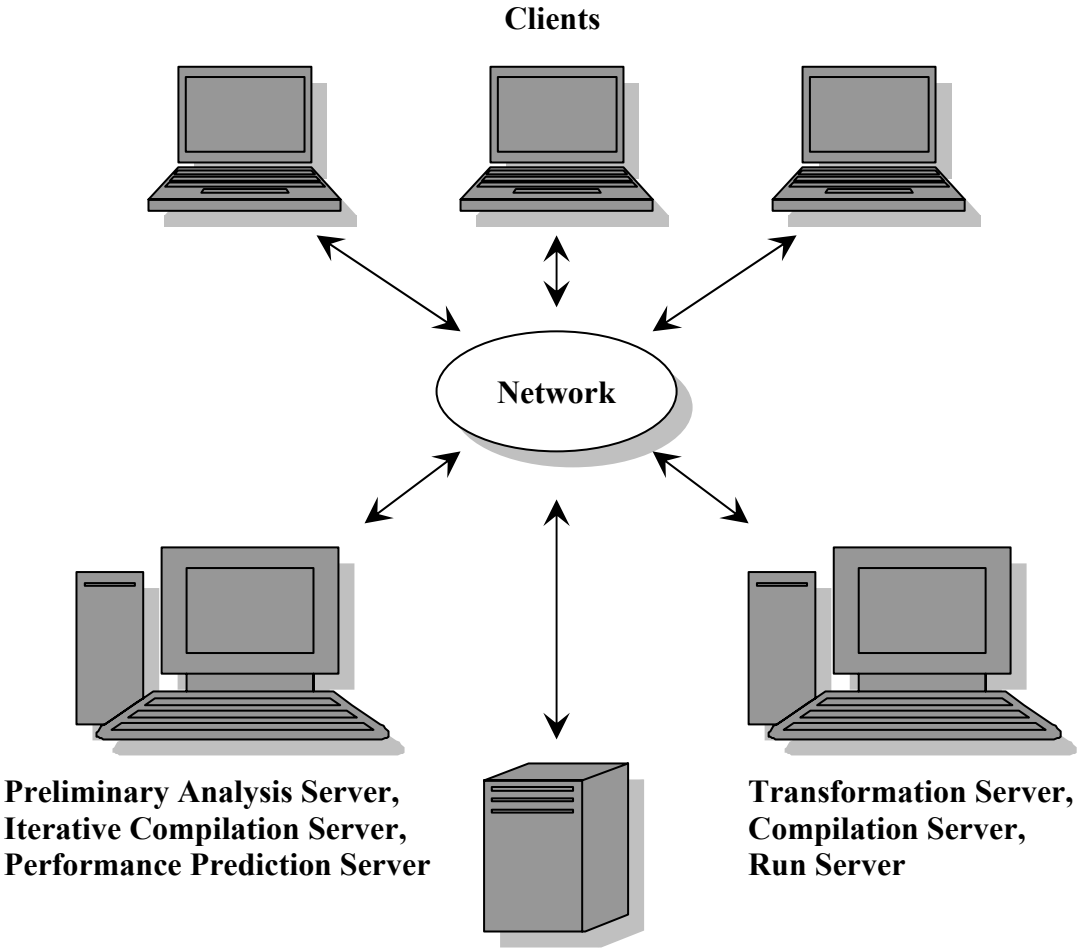


Figure 4.1: Software architecture of the optimising suite

This software architecture is flexible as it is easy to update the software and add new tools without the need to stop and restart all components. It supports auto error and fault recovery as servers can be restarted and the analysis and optimisation process can continue from the last correct state. The Run Server is introduced to ease portability between different platforms. This server is platform dependent and is used to execute applications and obtain their run-time parameters. Therefore, it is written in C to use low-level OS calls and recompiled for each platform using options specific for the particular platform. Most of the remaining software is written in Java and thus portable across platforms. Next in the software hierarchy are the Compilation and Transformation Servers. The Compilation Server sends requests to the Run Server to compile and execute programs and to collect various profile information. The Transformation Server supports array padding, loop tiling and loop unrolling transformations of Fortran programs, described in detail in section 3.1.

Finally, the Preliminary Analysis Server, Iterative Compilation Server and Performance Prediction Servers are tools that implement new analysis and optimisation techniques developed in this thesis. Briefly, the Preliminary Analysis Server is used for obtaining various preliminary information about the program such as the number of subroutines and loops, original execution time and so on, needed for further optimisations. The Iterative Compilation Server is used for applying new iterative compilation techniques. The Performance Prediction Server is used for predicting the ideal performance of the program. Similar architectures proved to be versatile and reliable in previous research projects such as MHAOTEU [ATA+00].

4.2.2 Platforms and applications

To demonstrate new platform-independent optimising techniques two distinctive, widespread platforms have been chosen for the experiments:

- Compaq Alpha 21264 500 MHz 512Mb, Digital Unix
- Intel Pentium III 650 MHz 256Mb, Windows 2000 Professional

For simplicity, further references to these platforms in the thesis will be as "Alpha" and "Pentium". The Alpha platform has a reduced instruction set (RISC) and the Pentium platform has a complex instruction set (CISC). Both platforms have a superscalar architecture with out-of-order execution support. Both platforms have two levels of cache: the Alpha has a 64KB 2-way set associative first level of cache and a 2MB direct-mapped second level of cache; the Pentium has a 16KB 4-way set associative first level of cache and a 256KB 8-way set associative second level of cache. These architectural features are described in detail in sections 2.1 and 2.2. More information about these platforms can be found in Appendix A.

Matrix multiplication (matmul), successive over relaxation (sor) and eight benchmarks from the SPEC'95 benchmark suite [SPE03] with reference datasets bigger than the cache size have been chosen for the experiments (two more benchmarks from this suite have been omitted due to technical compilation problems). Table 4.1 presents a brief description of each program (the SPEC suite description is taken from the official website [SPE03]) and shows the number of lines of the source code and the number of subroutines. All these applications are based on scientific numerical, floating-point algorithms within the scope of this

Application:	Lines of code/ Number of subroutines:	Description:
matmul	63 / 2	Matrix multiplication.
sor	59 / 2	Successive over relaxation method.
tomcatv	190 / 1	SPEC'95 FP. Fluid Dynamics / Geometric Translation. Generation of a two-dimensional boundary-fitted coordinate system around general geometric domains.
swim	429 / 6	SPEC'95 FP. Weather Prediction. Solves shallow water equations using finite difference approximations.
su2cor	2332 / 35	SPEC'95 FP. Quantum Physics. Masses of elementary particles are computed in the Quark-Gluon theory.
mgrid	484 / 12	SPEC'95 FP. Electromagnetism. Calculation of a 3D potential field.
applu	3868 / 16	SPEC'95 FP. Fluid Dynamics/Math. Solves matrix system with pivoting.
turb3d	2101 / 23	SPEC'95 FP. Simulation. Simulates turbulence in a cubic area.
apsi	7361 / 96	SPEC'95 FP. Weather Prediction. Calculates statistics on temperature and pollutants in a grid.
wave5	7764 / 105	SPEC'95 FP. Electromagnetics. Solves Maxwell's equations on a cartesian mesh.

Table 4.1: Description of applications

research and are written in Fortran. The two kernels, matmul and sor, are selected to allow simple and detailed performance evaluation and to analyse the developed techniques in depth. Their source codes are presented in figure 4.2. Their data sizes are selected to be larger than the cache size. The number of times these kernels are executed is selected in such a way that their execution times are similar to those of the SPEC benchmarks. The SPEC benchmarks are based on real applications that are hard to optimise. They are used to give a realistic and critical evaluation of the developed techniques. Moreover, all these programs are well studied ([WD98] and [MT99], for example) and can be used to compare results of the new techniques presented in this thesis with existing ones.

```
C     this subroutine is executed 8 times
      IMPLICIT REAL (A-F)
      PARAMETER (N=512)
      COMMON Y, A(N,N), B(N,N), C(N,N)
      DO I=1, N
       DO J=1, N
        DO K=1, N
         A(I,J)=A(I,J)+B(I,K)*C(K,J)
        END DO
       END DO
      END DO
```

(a) matmul

```
C     this subroutine is executed 256 times
      IMPLICIT REAL (A-F,X)

      PARAMETER (N=2048)
      COMMON Y, A(N,N)

      DO J=2, N-1
       DO I=2, N-1
        A(I,J)=A(I,J)+(A(I+1,J)+A(I-1,J)+A(I,J+1)+A(I,J-1))*0.00001
       END DO
      END DO
```

(b) sor

Figure 4.2: Source code of matmul and sor kernels

Before analysing and comparing various execution times, it is important to determine the precision of the timing on the particular platform. Execution time is generally measured using the system timer and can oscillate from run to run due to various operating system management processes. An additional tool has been created to measure the precision of the execution time. It executes the same time-consuming application 10 times and measures the deviation of the execution time. For the Alpha platform with Unix operating system, the obtained precision is 0.2 seconds. For the Pentium platform with Windows 2000 operating system, the original precision is 2.5 seconds. However, after setting the execution priority of the application just one level above normal, the precision becomes 0.5 seconds. Therefore, both platforms have a deviation less than 0.4% in execution time for all the programs used in the experiments.

4.3 Impact of program transformations

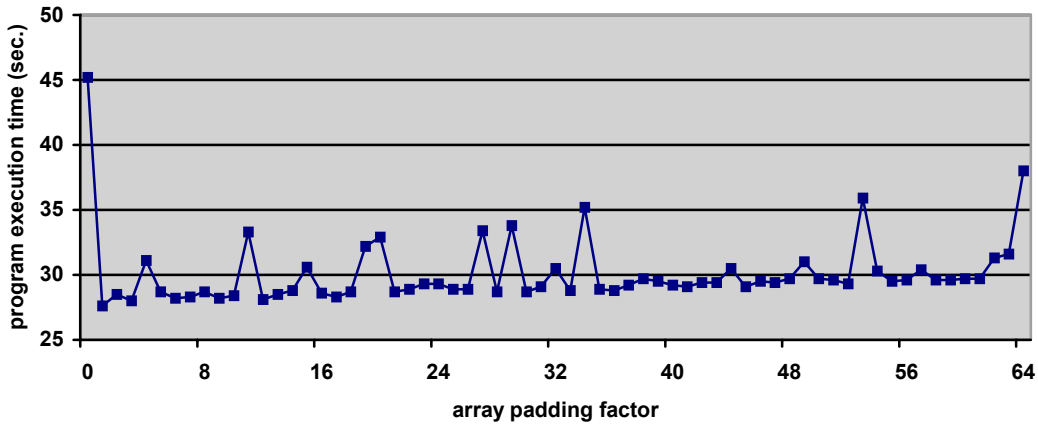
The aim of this section is to show the variable impact of program transformations on the program performance and to demonstrate why finding the best transformation parameter using known static and dynamic methods fails on modern platforms. It demonstrates that the impact of program transformations is of a non-linear nature and that it varies across different machines, underlining the challenge in developing portable automatic optimisation approaches. The impact of array padding, loop unrolling and loop tiling is examined using the small matmul kernel and the large swim benchmark on two platforms.

4.3.1 Array padding

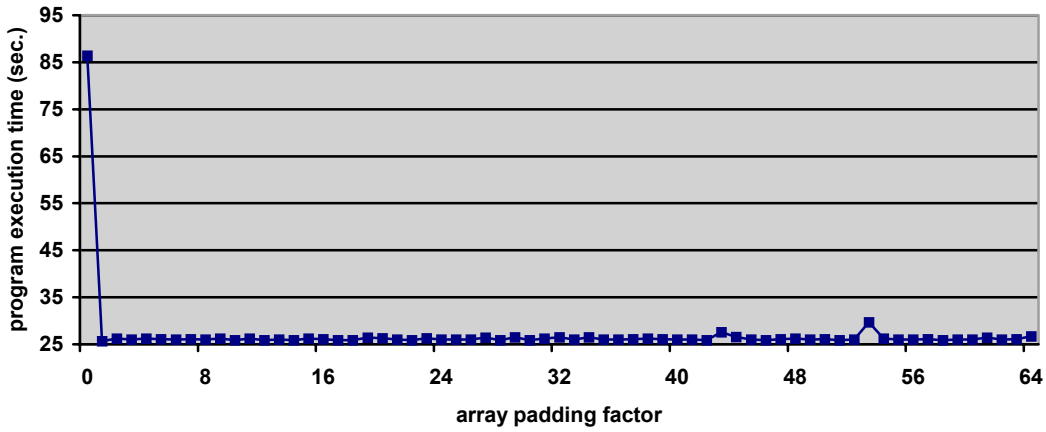
Several studies of the cache behaviour of various programs reviewed in detail in sections 3.1.3 and 3.2.3, show that many programs exhibit severe conflict misses, which degrade performance considerably. In such cases, intra- and inter-variable array padding is one of those program transformations that can be used to reduce conflict misses by inserting dummy data entries between the columns of arrays. Previous studies show a potentially large number of conflict misses in the cache behaviour of both matmul and swim programs. Therefore, array padding can be an effective transformation to improve their performance.

To simplify the experiments, intra-variable array padding is applied to all arrays simultaneously with the same parameter within the range of 1 to 64. This also changes the base address of each array, thus indirectly performing inter-variable array padding as well. Figure 4.3 demonstrates the changes in the execution time for matmul on the Alpha and Pentium platforms as a function of the array padding parameter and figure 4.4 presents the same experiments for swim on both platforms.

These experiments show that matmul and swim have indeed a large number of conflict misses that can be removed by array padding. This simple and effective transformation considerably improves program performance. It improves matmul performance by approximately 70% on the Pentium platform and by approximately 40% on the Alpha platform. Array padding improves the performance of the swim benchmark by approximately 45% on the Alpha platform and by approximately 25%



(a) Alpha platform

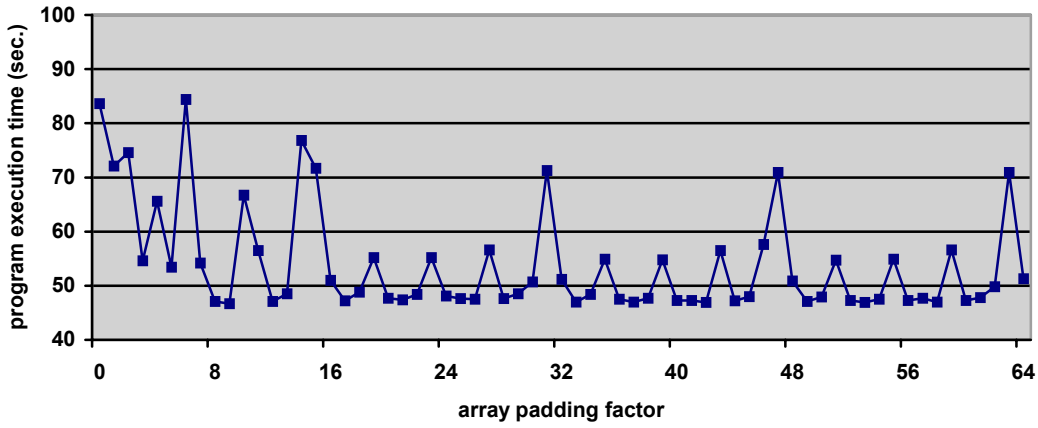


(b) Pentium platform

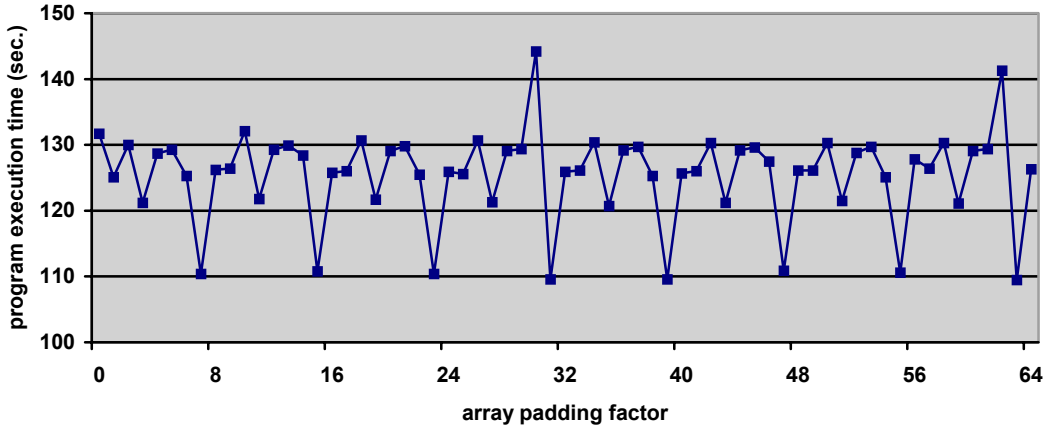
Figure 4.3: Execution time for varying array padding factors (matmul)

on the Pentium platform. These figures also show that the influence of array padding varies considerably across platforms. This is due to the fact that the effect of array padding depends on the cache organisation as shown in section 3.1.3. The oscillatory behaviour of array padding is due to the limited size and associativity of the cache. This means that array layouts and base addresses are changing in such a way after array padding, that they are mapped to the same cache lines periodically. Matmul has a higher performance improvement than swim because it performs less calculations per memory access and thus has a higher potential speed-up when cache misses are removed.

These experiments also explain why current static optimisation methods often fail to improve program performance after applying array padding. Static methods



(a) Alpha platform



(b) Pentium platform

Figure 4.4: Execution time for varying array padding factors (swim)

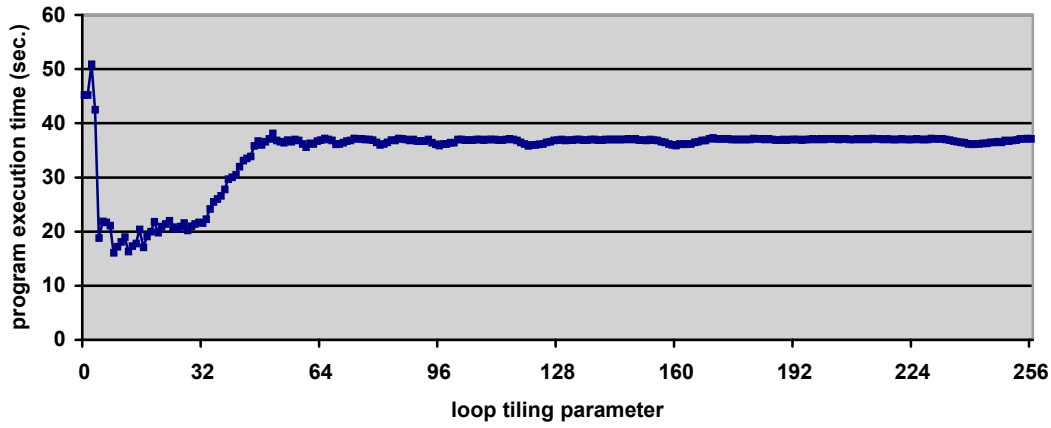
often use approximations to be tractable and may lack important run-time information such as array base addresses. Therefore, if these methods mispredict the array padding parameter even by a small factor in comparison with the best one, the overall performance can degrade considerably. For example, the difference in the execution times of the swim on the Alpha platform for the optimal padding factor 9 and for the following padding factor 10 is approximately 43%! The performance degradation for the same benchmark on the Pentium platform is also significant, approximately 14%, if padding factors 6 or 8 are selected that are close to the optimal padding factor 7.

4.3.2 Loop tiling

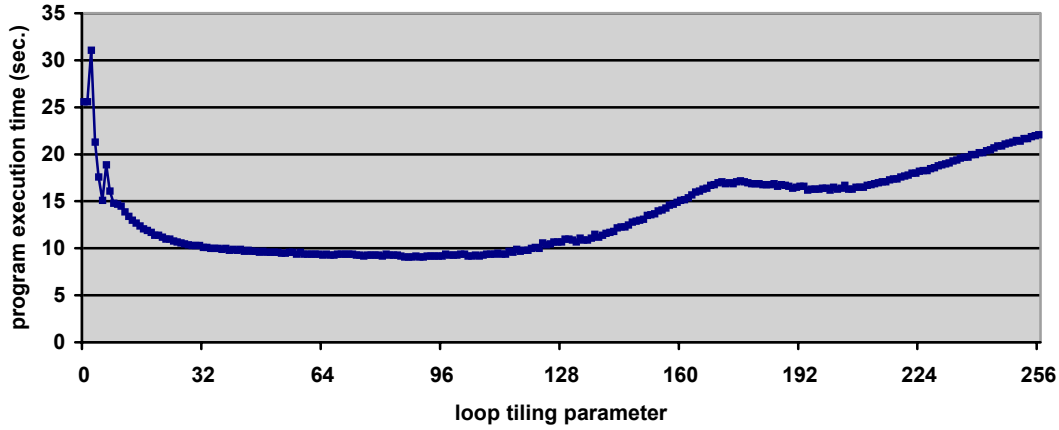
Loop tiling is used to improve cache reuse within a loop nest by dividing its iteration space into tiles as described in section 3.1.2. Various studies reviewed in section 3.2.2, show the effectiveness of this transformation in improving program performance when the data footprint of the original loop nest is bigger than the cache size. Since loop tiling changes the memory access pattern for the loop nest, it can also be used on its own or with array padding for removing conflict misses for this loop nest. Previous studies presented in section 3.2.3, introduce complex static techniques to analyse cache reuse for loop nests and to choose the best tile factor in such a way that tiles fit cache and exhibit minimum conflict misses.

Figure 4.5 shows the changes in the execution time for matmul on two platforms as a function of the loop tiling parameter. Analysis of these results shows that the original matrix multiplication algorithm for matrices bigger than the caches size has a poor locality, which can be improved using loop tiling. Graphs for both the Alpha and Pentium platforms have two distinct flat areas: approximately from 4 to 32 and from 46 to 255 for matmul on the Alpha platform, and from 32 to 122 and from 172 to 212 for this kernel on the Pentium platform. These results reflect the fact that both systems have two levels of cache and that tiles with the increasing size first fit level one cache and then fit level 2 cache. It is also possible to see some oscillations near the minimum in the graph for the Alpha platform in comparison with the relatively smooth graph for the Pentium platform. This can be explained by conflict misses occurring on the Alpha platform due to the limited associativity of its caches. The Pentium platform has a higher associativity of caches and therefore is capable of removing these misses at the hardware level.

Figure 4.6 shows the changes in execution time for the swim benchmark on two platforms as a function of the loop tiling parameter applied consecutively to the three most time consuming loops. As in the case of matmul, there are execution time oscillations near the minimum area for all three swim loops on the Alpha platform. However, loop tiling behaviour is different on the Pentium platform where one loop has multiple small oscillations while the other two loops have relatively smooth graphs. The loops in the swim benchmark perform more calculations per memory



(a) Alpha platform

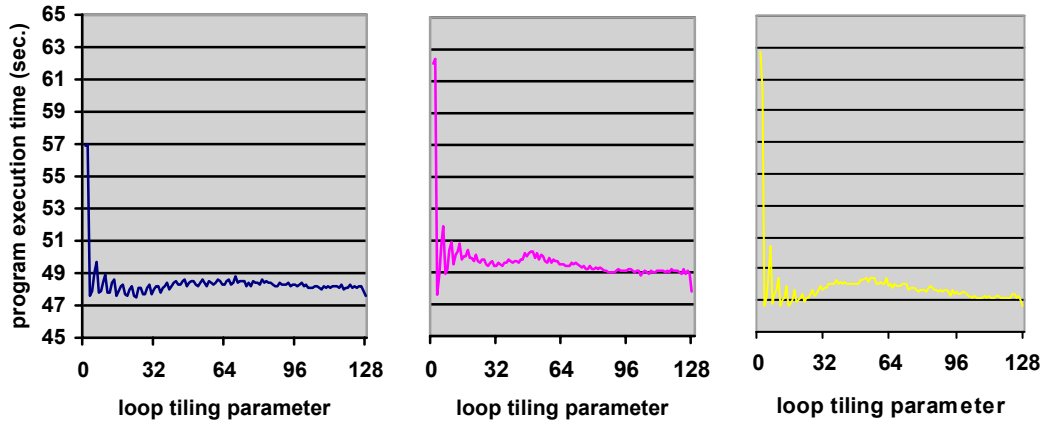


(b) Pentium platform

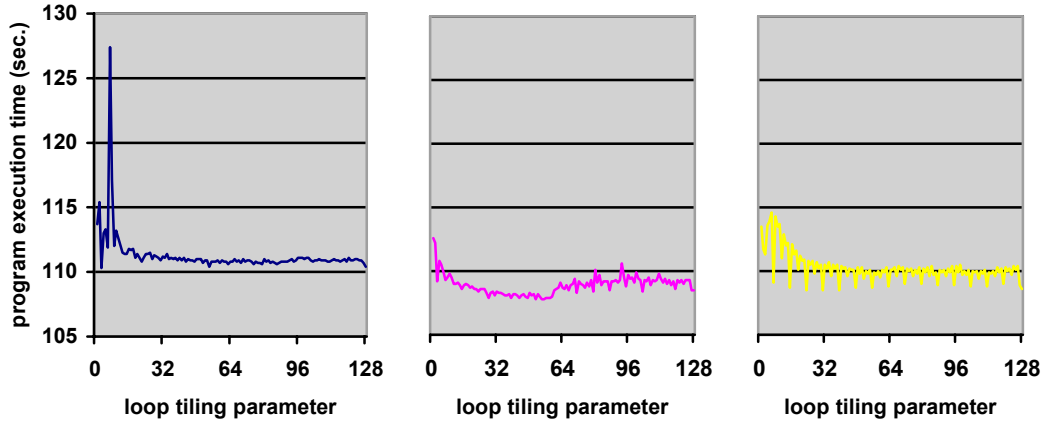
Figure 4.5: Execution time for varying loop tiling factors applied to the most time consuming loop (matmul)

access than matmul loops and therefore have less improvement after applying memory transformations.

Many static methods exist for analysing data locality and for choosing the best tile parameter as described in sections 3.2.2 and 3.2.3. They work reasonably well to eliminate capacity misses on simple kernels such as matrix multiplication. However, these methods encounter similar problems on complex programs with both conflict and capacity misses, as in the case of array padding. They lack the precision and run-time information to predict conflicts between memory accesses. These methods are evaluated in detail in section 6.5 where they are compared to the optimisation methods developed in this thesis.



(a) Alpha platform

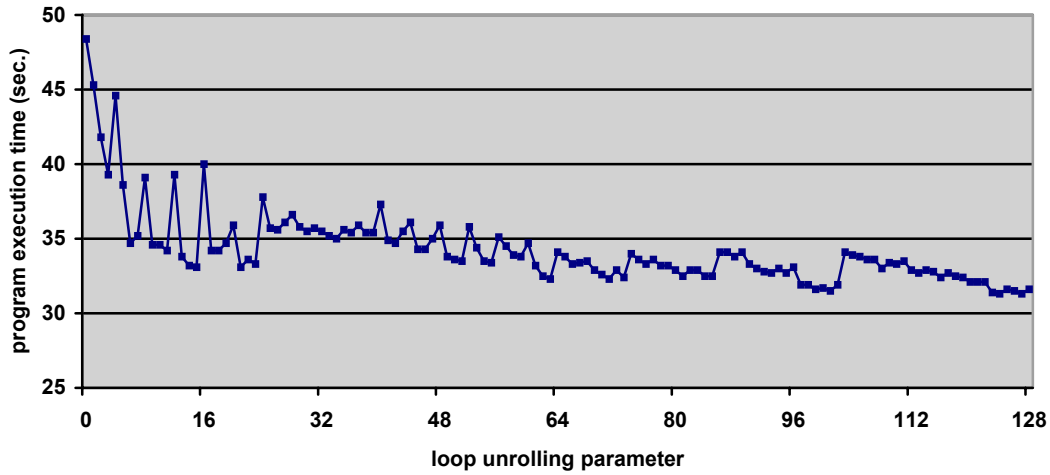


(b) Pentium platform

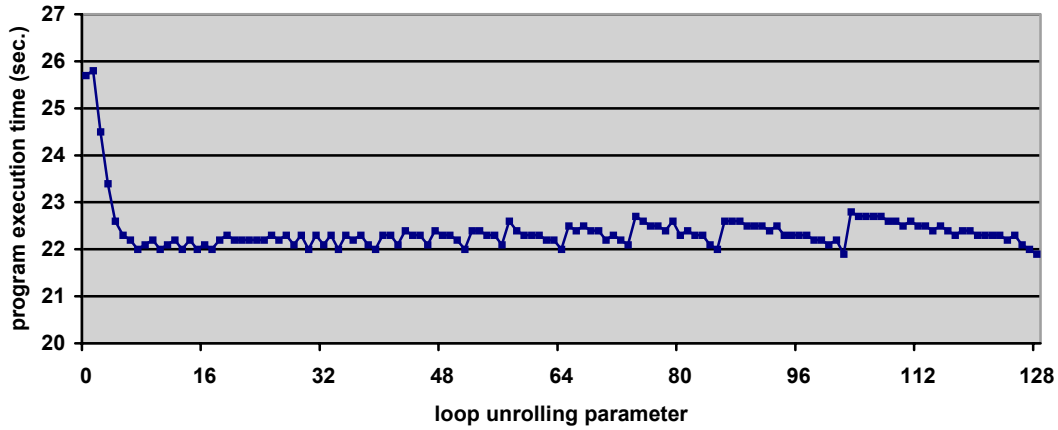
Figure 4.6: Execution time for varying loop tiling factors applied to the three most time consuming loops (swim)

4.3.3 Loop unrolling

Loop unrolling described in section 3.1.4 is used to improve ILP by increasing the number of operations within a single loop iteration, and to improve data locality by reducing the number of memory accesses through better register reuse. On the other hand, loop unrolling increases the size of the code that may result in performance degradation if the transformed code is larger than the instruction cache size. Naturally, as the code grows, the effect of loop unrolling depends on the number of available registers and on other hardware resources. Thus, high loop



(a) Alpha platform

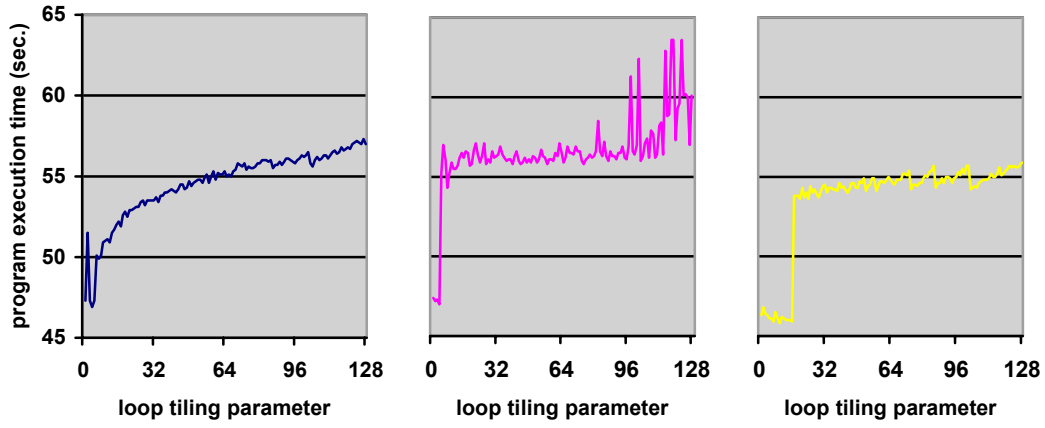


(b) Pentium platform

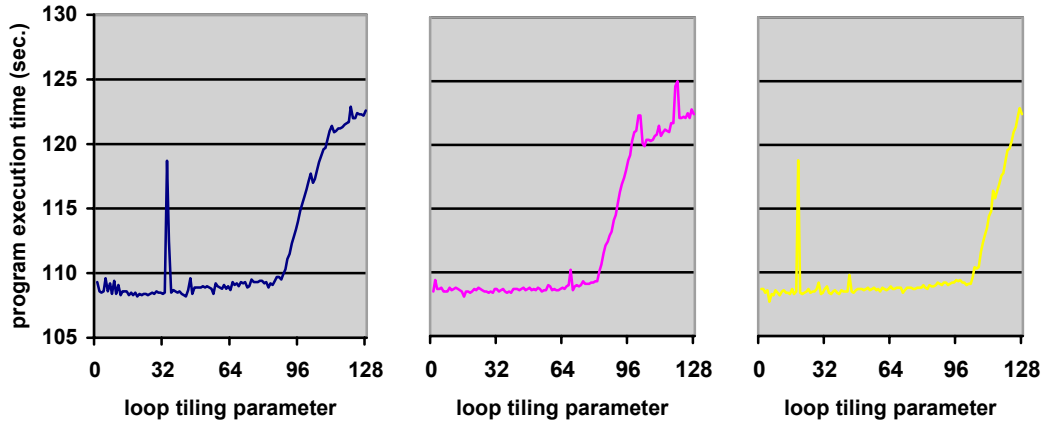
Figure 4.7: Execution time for varying loop unrolling factors applied to the most time consuming loop (matmul)

unrolling factors can degrade the performance of large loops, and small unrolling factors are generally beneficial as shown in studies presented in section 3.2.1.

Figures 4.7 show the changes in the execution time for matmul on two platforms as a function of the loop unrolling parameter within the range of 2 to 128. Both graphs have similar behaviour. The execution time decreases rapidly on both graphs for small unrolling factors until 15 on the Alpha platform and until 7 on the Pentium platform. Further, performance improvements slow down and are negligible on both platforms. This shows that after some unrolling factor threshold, all hardware resources are utilised and all the potential ILP is exploited. These graphs also show that loop unrolling either improves performance or at least does not degrade it for the



(a) Alpha platform



(b) Pentium platform

Figure 4.8: Execution time for varying loop unrolling factors applied to the three most time consuming loops (swim)

kernel within the chosen range of unrolling factors. This is explained by the fact that matmul is a small and simple kernel with a high register reuse. Therefore, the transformed code fit the instruction cache for all chosen unrolling factors.

The behaviour of loop unrolling is different for larger loops. This situation is demonstrated in figure 4.8 that presents graphs with changes in execution time for the large swim benchmark on two platforms as a function of the loop unrolling parameter. Loop unrolling is applied to the three most time consuming loops of this benchmark. There are small improvements in the execution time after applying loop

unrolling with small factors for all three loops on the Alpha platform with a sharp performance degradation after a certain parameter, which is 6 for the first loop, 4 for the second loop and 16 for the last loop. The little improvements in this benchmark performance in comparison with matmul is explained by the fact that the body of the original loop is already large enough to exploit ILP and there is a little potential for further improvements. The sharp performance degradation shows that the transformed code becomes larger than the instruction cache size after a particular unrolling factor. There is a less sharp performance degradation on the Pentium platform after applying loop unrolling with large factors and is explained by the fact that this platform has a complex instruction set. Therefore, both the original and transformed programs are considerably more compact than on the Alpha platform and fit the instruction cache for larger unrolling factors.

The oscillations that can also be seen in both the matmul and swim graphs depend on many factors, for example the way the compiler allocates registers and the way the processor executes instructions and predicts branches. These oscillations demonstrate that choosing a fixed unrolling factor or using static techniques for predicting ILP generally improves the performance for small kernels but may not be optimal or can even degrade performance for large programs such as the SPEC benchmarks.

The experiments presented in this section show a high potential for improving the program performance using array padding, loop tiling and loop unrolling. It also demonstrates why modern static and dynamic optimisation techniques often fail to deliver this performance improvement. The following section presents a new feedback-directed optimisation method that considerably outperforms the state-of-the art compilers with no architectural knowledge.

4.4 Basic search strategy

The main objective of a compiler optimisation strategy is to decide which transformations to apply. It is usually guided by information obtained using static or dynamic analysis and heuristics that reduce the transformation space considered. The majority of existing research in optimisation via high level restructuring relies on static information and often fails to achieve the best performance due to the

1. *profile original program*
2. *choose set of arrays and loops*
3. *apply data transformations:*
   - *apply array padding (**1..N_a**) for all global arrays*
   - *run program variant and record the best execution time*
   - *select the best transformation (minimal execution time)*
4. *apply loop transformations:*

for each selected loop nest:

for each loop from this nest:

if loop is not innermost and is within a perfect nest:

- *apply loop tiling (**2..N_t**) for the loop nest*
- *run program variant and record the best execution time*

if loop is innermost:

- *apply loop unrolling (**2..N_u**) for the innermost loop without tiling*
- *run program variant and record the best execution time*

if the best tiling factor is found for the enclosing iterators within the loop nest:

- *choose best tiling transformation*
- *apply loop unrolling (**2..N_u**) for the innermost loop*
- *run program variant and record the best execution time*

select the best transformation for the loop nest (either loop unrolling or a combination of both loop tiling and loop unrolling)

Figure 4.9: Basic search strategy algorithm

imprecision of models, as described in sections 3.1 and 4.3. Furthermore, due to a highly erratic behaviour of each transformation, determining the best combination for an arbitrary program and platform is very difficult. To overcome this problem, the approach presented in this thesis primarily deals with developing search-based iterative compilation techniques that are solely based on dynamic information and have minimal or no architectural knowledge at all.

Ideally, iterative compilation is a process that creates multiple variants of a program for all possible transformations, executes them and chooses the one with the

best performance. However, the transformation search space for real programs can be overwhelmingly large making it impossible to investigate within a reasonable time.

For instance, consider the swim benchmark from the SPEC suite with 14 arrays and 8 double-nested loops, and only three transformations: array padding with parameters up to 64, loop tiling with parameters up to 256, and loop unrolling with parameters up to 128. The search space for this benchmark consists of approximately 10^{52} possible different transformations that is unrealistic to explore. An additional problem is that the same dataset has to be used during iterative compilation. This means that the best variant of the program found during iterative compilation for the particular dataset may not be optimal if dataset size or content are changed. A potential solution is to optimise a program several times for some typical datasets with the most time consuming branches taken and to embed the conditional checks on the dataset into the final program to choose different optimised versions. However, this is out of the scope of this thesis. Preliminary results of using smaller datasets for iterative compilation, shown later in section 6.6, demonstrate such a possibility. However, this is out of the scope of this thesis. Therefore, the datasets of the programs studied in this thesis have a fixed size and the influence of their content on the optimisation process is a subject for future research.

Figure 4.9 presents a new basic search strategy algorithm that reduces the search space dramatically by considering data and loop transformations separately one by one instead of all combinatorial options. Initially, the program is profiled and those subroutines that dominate execution time are marked. Only loop nests and arrays referenced within these subroutines are selected for the search strategy to remove unimportant loops from further investigation.

As data transformations are global in effect, they are considered first on the assumption that local loop transformations can later compensate for some adverse effects that can be caused locally by the global data transformations. First, array padding is applied to the first dimension of the marked arrays. If there are N_a padding factors to consider and m arrays, then the number of different padding combinations is $N_a{}^m$. To reduce this complexity, each array is padded with the same factor, reducing the number of iterations to N_a. For each array padding parameter, a

new variant of the program is executed and the best padding factor, according to the minimal execution time, is selected.

When the process of choosing the array padding factor is completed, the best array padding transformation is incorporated in all further program variants. After that, loop transformations are applied sequentially. For each loop from a selected loop nest, if this loop is not innermost and is within a perfect loop nest then loop tiling is applied with factors from ***2*** to N_t. Each new variant of the program is executed and the best execution time is recorded. When the loop is innermost, loop unrolling is applied first with factors from ***2*** to N_u. Each new variant of the program is executed and the best execution time is recorded. Further, according to studies presented in chapter 3, applying a combination of loop tiling and loop unrolling can potentially achieve better performance improvements than after applying each transformation on its own. Therefore, if the best loop tiling factor has been found for the outer enclosing iterators within this loop nest, the loop tiling with this parameter is applied and loop unrolling is further applied for the innermost loop with factors from ***2*** to N_u. Each new variant of the program is executed and the best execution time is recorded. Finally, the best sequence of transformations, which is either loop tiling or loop unrolling or a combination of both, is selected for this loop nest to be used with the following transformations.

This basic optimisation strategy considerably reduces the search space. For example, the search space for the swim benchmark from the SPEC suite mentioned above is reduced from approximately 10^{52} variants to approximately 2500 possible variants. Though this number is still high, it can be tolerable for small programs and kernels with a long lifetime that need to be well optimised, thus making this approach a realistic alternative to other optimisation methods. However, this basic strategy treats data and loop transformations, which may potentially influence each other, separately and therefore may not achieve the best possible performance. Nevertheless, the following evaluation section shows that this simplified strategy can still achieve considerable performance improvements without architectural knowledge compared to current compiler static and feedback-directed optimisation techniques.

4.5 Experimental results

Development of any new optimisation technique should ideally be compared with methods implemented in the best commercial optimising compilers. The following optimising compilers are chosen for the experiments:

- Digital Fortran 5.2 (Alpha platform)
- Intel Fortran 6.0 (Pentium platform)

Both compliers support static data and loop transformations. However, since static optimisation methods can fail to achieve the best performance on rapidly evolving hardware or may even degrade it, dynamic methods are used as well. Both of the above compilers support feedback-assisted compilation. Briefly, it consists of three steps: program instrumentation where special code is inserted into the program to obtain run-time information, execution of the instrumented code to collect this run-time information, and finally feedback-assisted compilation where the program is optimised using run-time information.

For further reference and comparison all applications are compiled using three options:

Opt.1) maximum internal optimisations with data and loop transformations disabled;

Opt.2) maximum internal optimisations with data and loop transformations enabled;

Opt.3) feedback-assisted optimisations.

This allows comparison of the best static and dynamic optimisation methods implemented in the state-of-the-art compilers with the new techniques developed in this thesis. It also allows one to analyse the influence of static compiler data and loop transformations on the program performance.

The execution times for 2 kernels and 8 SPEC benchmarks used in the experiments with the optimisations described above for the Alpha and Pentium platforms are presented in table 4.2 with the best execution times highlighted. Figure 4.10 present graphs with execution time improvements of Opt.2 and Opt.3 over Opt.1 on both platforms, where improvement is calculated as

$\frac{Tnew - Toriginal}{Toriginal} \cdot 100\%$. These results support the statement made in the previous sections that current static and dynamic optimisation techniques with data and loop transformations are still not efficient and may even degrade performance. On the Alpha platform, internal static compiler optimisations are only capable of achieving a considerable performance improvement, approximately 30%, on matmul when applying data and loop transformations. Swim, su2cor, applu and wave5 have performance improvements between 10 and 15% after loop and data transformations; sor, mgrid and apsi have a negligible performance improvement; tomcatv has its performance slightly degraded and finally, turb3d has its performance degraded considerably by 20%. Feedback-assisted optimisations perform better only on mgrid and wave5 on the Alpha platform. For all other codes, dynamic optimisations fail to improve on static optimisations on the Alpha platform. On the Pentium platform, the influence of both static and dynamic optimisations on program performance is insignificant. Similar to the Alpha platform, some codes have performance improvements while others has their performance degraded, but in all cases, the change in the execution time is less than 3.5% of the original time. Furthermore, feedback-directed optimisations slightly degrade the performance of mgrid and apsi. These results for the Pentium platform can be explained by its smaller cache size, lower memory throughput and higher instruction latencies common for CISC platforms as described in section 2.1.2. Furthermore, Digital Fortran for the Alpha platform has an aggressive optimisation engine and therefore can achieve better performance.

The basic search strategy is evaluated on the same programs on the Alpha and Pentium platforms. All applications are first profiled to choose the subroutines that dominate execution time. Within each chosen subroutine, all loop nests and arrays referenced are selected for the use in the basic search strategy. The maximum array padding factor has been chosen as N_a = 64. The maximum loop tiling and unrolling factors have been chosen as N_t = 512 and N_u = 512. To compare the efficiency of the search strategy with the best commercial compilers, table 4.3 presents execution time improvements achieved after applying iterative compilation with the basic search

	Alpha platform			Pentium platform		
Application:	**Opt.1**	**Opt.2**	**Opt.3**	**Opt.1**	**Opt.2**	**Opt.3**
matmul	45.2	**31.1**	**31.1**	86.4	**83.9**	85.0
sor	48.4	48.4	48.4	48.9	48.9	48.9
tomcatv	**79.9**	83.1	82.0	144.8	144.3	**140.6**
swim	83.6	**71.4**	**71.4**	**131.7**	132.5	131.8
su2cor	79.5	**69.6**	70.1	170.1	169.6	**168.8**
mgrid	85.3	84.6	**80.1**	189.0	**188.9**	191.1
applu	93.5	**83.0**	84.3	170.2	166.6	**164.4**
turb3d	**137.1**	163.6	150.1	190.5	**188.7**	189.3
apsi	62.0	**60.1**	60.3	111.0	**110.2**	113.5
wave5	73.5	65.3	**62.5**	121.3	119.3	**119.0**

Opt.1) maximum internal optimisations with data and loop transformations disabled:
"-O4" for the Alpha platform,
"/O2 /Qunroll0" for the Pentium platform;

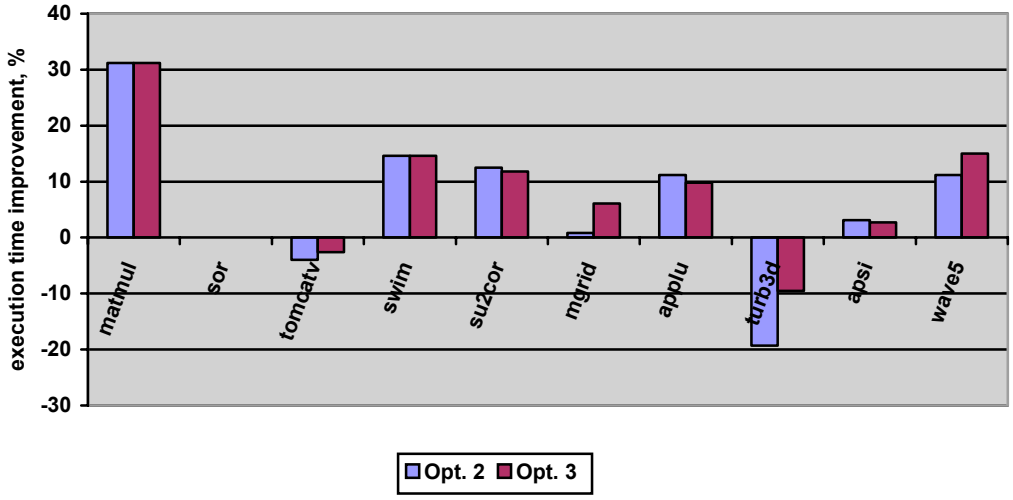
Opt.2) maximum internal optimisations with data and loop transformations enabled:
"-O5" for the Alpha platform,
"/O3 /Qunroll" for the Pentium platform;

Opt.3) feedback-assisted optimisations:
"-O5 -feedback" for the Alpha platform,
"/O3 /Qunroll /Qprof_use" for the Pentium platform.

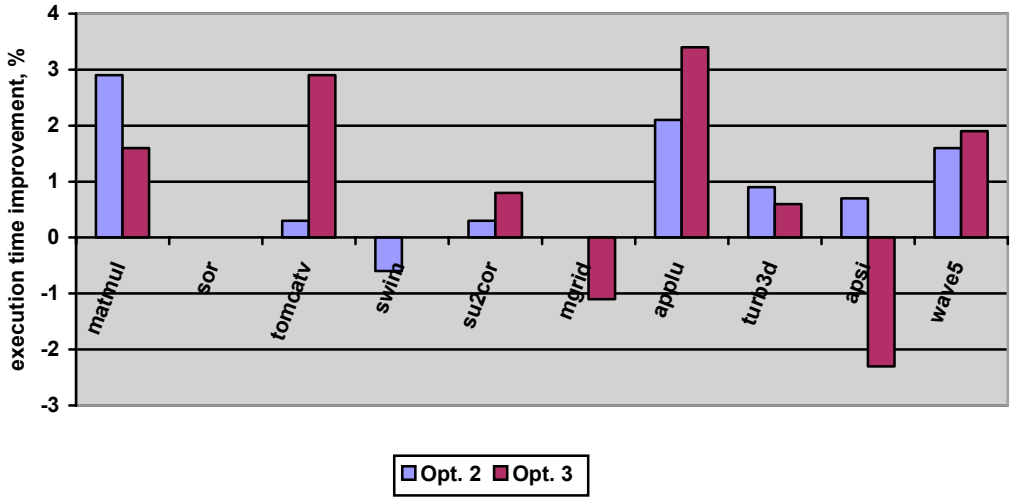
Table 4.2: Application execution times after internal compiler optimisations (best times are highlighted)

strategy relative to Opt.1, Opt.2 and Opt.3, and figure 4.11 presents a graph with these results.

Since iterative compilation selects only the best variants of the transformed program, it achieves performance improvement in all cases unlike best static and feedback directed optimisation methods that may degrade performance of some codes. Moreover, iterative compilation achieves high performance improvements on small kernels that have relatively few loops and arrays after several thousands of iterations. The original matmul has a poor data reuse that can be dramatically



(a) Alpha platform



(b) Pentium platform

Figure 4.10: Execution time improvements (%) of Opt.2 and Opt.3 over Opt.1

improved using padding, tiling and unrolling as described in various studies presented in chapter 3. After iterative compilation, matmul achieves a considerable performance improvement of 80.1% on the Alpha platform over Opt.1 and an even higher improvement of 92.6% on the Pentium platform. Sor has better locality and therefore less potential for improvement after memory optimisations. Nevertheless, this kernel still achieves a considerable performance improvement of 28.6% on the Alpha platform after iterative compilation and 16.0% on the Pentium platform. On average, both kernels achieve around 54% performance improvement on both platforms after 1599 iterations over Opt.1. These kernels achieve considerable

performance improvements of 50% on the Alpha platform and of around 54% on the Pentium platform on average even over Opt.2 that are high loop and data optimisations and over Opt.3 that are feedback directed optimisations. Such high improvements after iterative compilation are due to simple loop structures of such kernels that allow easy, straightforward and efficient memory optimisations of the code. However, this may not be the case for real large applications with multiple loop nests where data reuse occurs across nests [MT99]. In such cases, transforming loop nests separately can potentially reduce overall optimisation effect. Nevertheless, results presented in table 4.3 and figure 4.11 for eight SPEC'95 benchmarks show that iterative compilation with the basic search strategy is capable of achieving high performance improvements even on real complex applications with multiple loop nests.

Improvements vary considerably across applications and platforms. For the Alpha platform, performance improvements vary between 13.0% and 45.1% and for the Pentium platform between 4.8% and 22.5%. The number of iterations needed for the optimisation varies between 5694 and 27180 for both platforms as the same number of loops and arrays has been selected for simplicity. It should be noted that the higher number of iterations means that more loops have been selected for the optimisations. However, it does not necessarily mean that the achieved performance improvement is higher as naturally not all loops can benefit from memory optimisations. For example, swim has the highest performance improvement of 45.1% over Opt.1 among all other SPEC benchmarks on the Alpha platform after only 6205 iterations and one of the highest performance improvements of 18.0% on the Pentium platform. On the contrary, applu has one of the lowest performance improvements of 16.0% among SPEC benchmarks on the Alpha platform after a considerable 27180 iterations and the lowest improvement of 4.8% on the Pentium platform. This can be explained by the fact that swim has only three most time consuming loops with a simple structure operating with large two-dimensional matrices that can be easily transformed and can benefit the most from memory optimisations. In contrast, applu has several time consuming loops with either complex structures or non-perfectly nested loops operating on five-dimensional matrices and are difficult to transform.

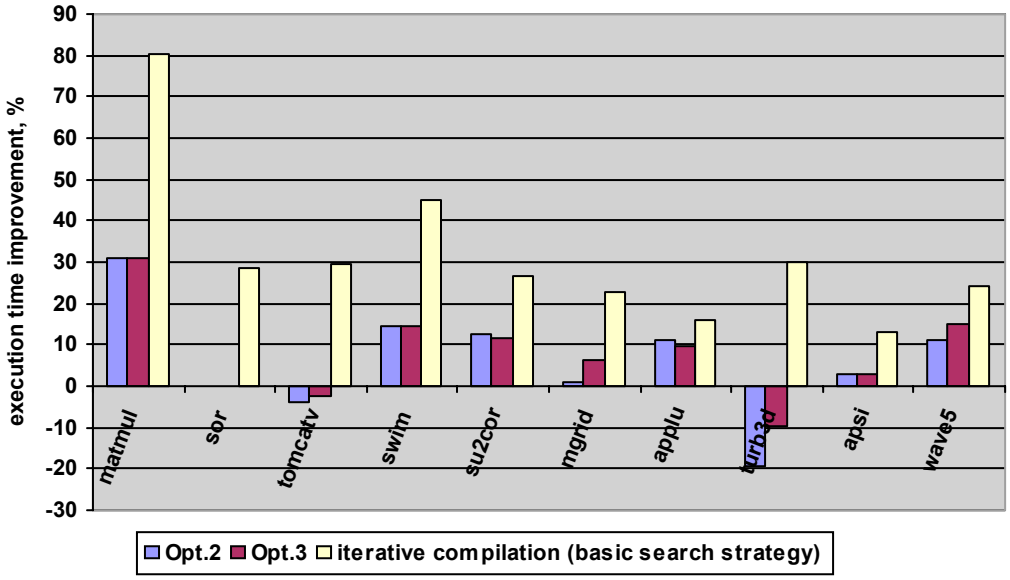
Application:	Number of iterations:	Execution time improvements:		
		Over Opt.1	Over Opt.2	Over Opt.3
matmul	1599	80.1%	71.1%	71.1%
sor	1599	28.6%	28.6%	28.6%
average (kernels)	1599	54.4%	49.9%	49.9%
tomcatv	7738	29.6%	32.3%	31.4%
swim	6205	45.1%	35.7%	35.7%
su2cor	9280	26.5%	16.0%	16.7%
mgrid	14905	22.5%	21.9%	17.5%
applu	27180	16.0%	5.4%	6.9%
turb3d	5694	30.1%	41.4%	36.2%
apsi	10813	13.0%	10.2%	10.6%
wave5	7744	24.2%	14.6%	10.8%
average (benchmarks)	11195	25.9%	22.2%	20.7%

(a) Alpha platform

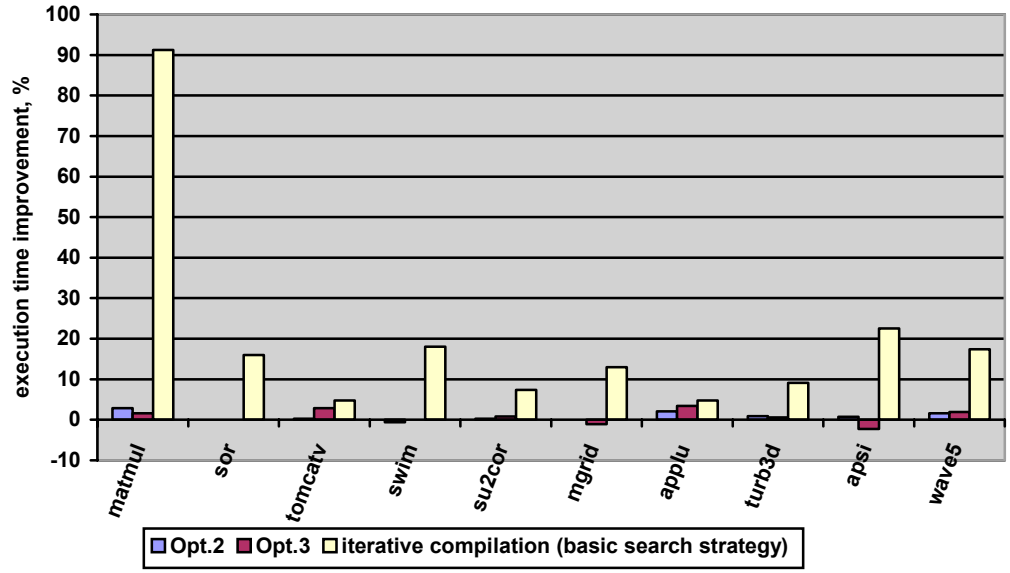
Application:	Number of iterations:	Execution time improvements:		
		Over Opt.1	Over Opt.2	Over Opt.3
matmul	1599	92.6%	92.4%	92.4%
sor	1599	16.0%	16.0%	16.0%
average (kernels)	1599	54.3%	54.2%	54.2%
tomcatv	7738	4.8%	4.5%	2.0%
swim	6205	18.0%	18.5%	18.0%
su2cor	9280	7.4%	7.1%	6.7%
mgrid	14905	13.0%	13.0%	13.9%
applu	27180	4.8%	2.8%	1.5%
turb3d	5694	9.1%	8.3%	8.6%
apsi	10813	22.5%	22.0%	24.2%
wave5	7744	17.4%	16.1%	15.8%
average (benchmarks)	11195	12.1%	11.5%	11.3%

(b) Pentium platform

Table 4.3: Execution time improvements (%) after iterative compilation with the basic search strategy over Opt.1, Opt.2 and Opt.3



(a) Alpha platform



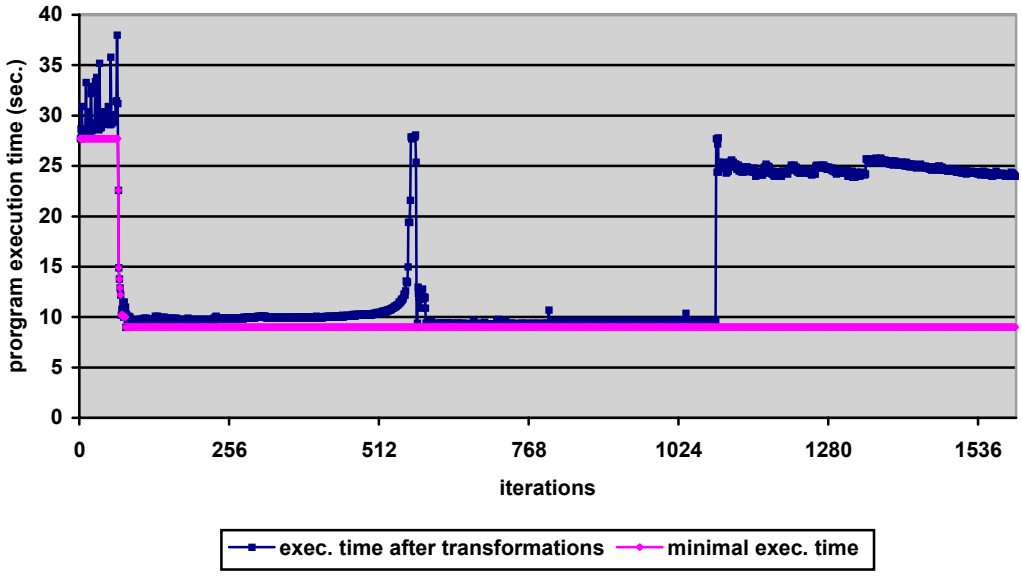
(b) Pentium platform

Figure 4.11: Execution time improvements (%) after iterative compilation with the basic search strategy, Opt.2 and Opt.3 over Opt.1

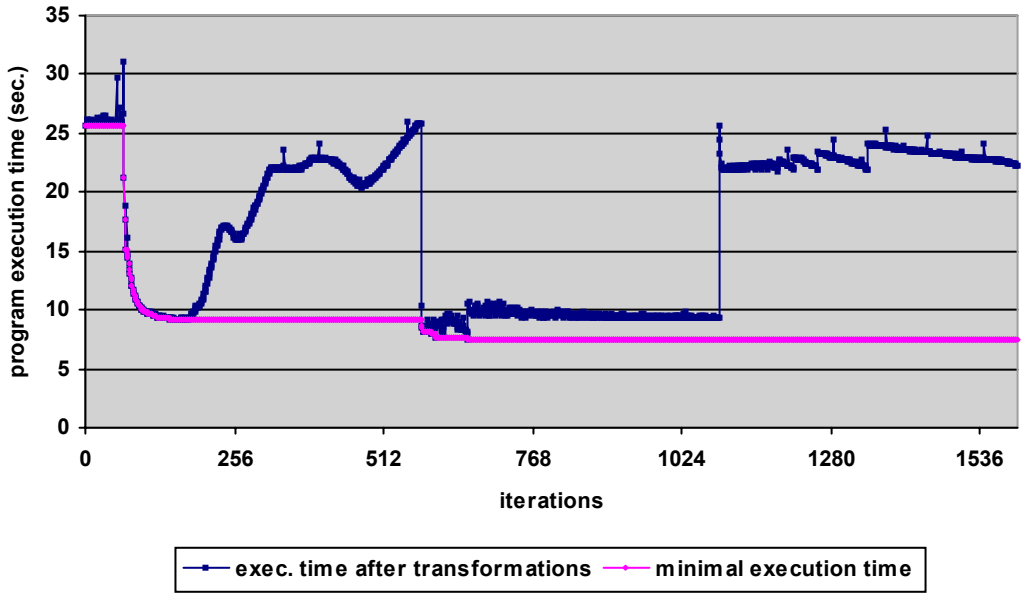
Naturally, the outcome of iterative compilation as well as the performance improvements after static or dynamic optimisations depends heavily on the processor architecture, memory hierarchy and compiler technology used as shown in chapter 3 and in section 4.3 and therefore can vary considerably across different platforms. The results presented in table 4.3 for SPEC benchmarks demonstrate this statement. For example, tomcatv, su2cor and turb3d have considerable performance improvements

on the Alpha platform of around 30% whilst the same benchmarks have relatively small improvement of around 5 to 9% on the Pentium platform. Apsi, on the other hand, achieves better performance improvement of 22.5% on the Pentium platform than on the Alpha platform where its improvement is smaller of 13.0%. Finally, mgrid and wave5 have high performance improvements on both platforms. Such variations are explained by differences between CISC and RISC architectures of the used platforms and by differences in memory hierarchy: the Alpha platform has larger caches but with less available associativity than the Pentium platform.

When compared to Opt.2 that are static compiler loop and data optimisations and Opt.3 that are feedback-directed optimisations, iterative compilation should achieve less performance improvements. The above experiments for both kernels and SPEC benchmarks prove this statement. Iterative compilation achieves 54.4% improvement for kernels on average over Opt.1 and 49.9% over Opt.2 and Opt.3 on the Alpha platform. It achieves 54.3% improvement for the same kernels on average over Opt.1 and 54.2% over Opt.2 and Opt.3 on the Pentium platform. SPEC benchmarks have performance improvements of 25.9% on average over Opt.1, 22.2% over Opt.2 and 20.7% over Opt.3 on the Alpha platform and performance improvements of 12.1% over Opt.1, 11.5% over Opt.2 and 11.3% over Opt.3 on the Pentium platform. These figures show that though performance improvements after iterative compilation with the basic search strategy are slightly smaller for Opt.2 and Opt.3 than Opt.1, overall they are considerable for all kernels and for most of the benchmarks. The smaller average performance improvements on the Pentium platform in comparison with the Alpha platform are explained by the limitations of optimisations on CISC platform due to higher instruction latencies and due to smaller cache size and lower memory throughput. Nevertheless, these results are remarkable, considering that this search strategy does not have knowledge of the underlying hardware and has a minimal knowledge about the program structure. This is important for optimising programs on rapidly evolving hardware since performance improvement varies considerably from one platform to another. Besides, it shows the high potential for performance improvements that modern static and dynamic methods fail to explore. The following chapters will compare the obtained results with other methods and will present a new realistic approach for predicting best potential performance for programs.



(a) Alpha platform



(b) Pentium platform

Figure 4.12: Changes in execution time during each iterative step (matmul)

To demonstrate the iterative compilation process in detail, figure 4.12 shows the execution time of the transformed matmul kernel on the Alpha and Pentium platforms during each iterative step and the best achieved execution time. Matmul has a single subroutine and a triple-nested loop referencing three arrays. All three arrays have been selected for iterative compilation. However, only the inner and outer loops have been selected for iterative compilation for the sake of simplicity and

to reduce the number of iterations. Therefore, iterative compilation for this kernel consisted of four major steps:

- global array padding (1..64);
- loop tiling of the outer loop (65..576);
- best loop tiling of the outer loop plus loop unrolling of the inner loop (577..1088);
- loop unrolling of the inner loop (1089..1599),

where numbers in brackets show iterations that belong to each step. The above graphs demonstrate that optimisations depend heavily on the hardware and that best transformation factors vary considerably across platforms. For example, the best performance for matmul on the Alpha platform is achieved using array padding and loop tiling, whilst on the Pentium platform it is achieved using array padding and the combination of loop tiling and unrolling. This also demonstrates the difficulties which static optimisation methods face, as these techniques should not only consider separate transformations but also their combinations.

The basic search strategy presented in this chapter has a major drawback: the time spent for program optimisation is considerably higher that used by modern static or dynamic approaches – it can require thousands of runs of program variants. This can be acceptable and useful for optimising small programs and kernels whose lifetime is greater than the overall optimisation time, but in many other cases the iterative compilation time is unacceptable. To overcome this drawback, the following chapters will investigate the possibilities in finding the trade-off between the speed-up and the iterative compilation time by reducing the search space and by using advanced search strategies.

4.6 Summary

This chapter shows the influence of array padding, loop tiling and unrolling on application performance, and describes a new iterative optimisation approach including all these transformations that outperforms current static and feedback directed compiler techniques. This approach allows optimisers to adapt to any new platform by way of feedback directed iterative compilation. Considerable speed-up

has been achieved after applying iterative compilation for two well-known kernels and eight SPEC FP benchmarks across two platforms compared to the results obtained using native high-level restructurers and platform-specific profile-directed optimisers that employ the same transformations. Furthermore, iterative compilation never degrades program performance unlike current static and dynamic methods that can considerably degrade it.

This chapter shows that it is possible to dramatically outperform current static and dynamic optimisation methods using iterative compilation with the same or a smaller set of transformations, regardless of the platform. The major drawback of the new approach is a very large number of iterations making it very time-consuming. To overcome this problem the next chapters investigate new techniques to predict the possible performance improvements before applying costly iterative compilation and describe new search strategies to dramatically reduce the search space.

Chapter 5

Performance Prediction

This chapter describes a new fast and accurate technique that can predict the potential benefit from applying memory transformations to various program sections. Since the optimisation process can be tedious and time-consuming, this technique allows the removing of those program sections from the optimisation process that do not have the potential for performance improvement. It is particularly important for iterative compilation where investigating only one loop nest may require thousands of executions of program variants. This technique is platform-independent and transforms the assembler code of the original program so that the new program variant does not exhibit cache misses. Thus, profiling the original and new program followed by a comparison of execution times provides a fast evaluation of the potential benefit from applying memory transformations that target cache misses. The advantages of this technique over existing ones are compared at the end of this chapter. Chapter 6 shows how this performance prediction technique effectively reduces the search space.

5.1 Introduction

In performance critical applications, memory latency is frequently the dominant overhead and in many cases, automatic compiler-based optimisations to improve memory performance are limited. As shown in the previous chapter, in the majority of such cases iterative compilation provides a significant performance improvement. However, this method is excessively time-consuming and is therefore unrealistic to use in general purpose computing. Furthermore, as the potential benefit from optimisation is unknown there is no way to judge the amount of effort worth spending and there are no criteria from which to decide when the optimisation process should stop, i.e. when the optimal memory performance has been achieved or sufficiently approached.

This leads to the following technical question: is it possible to estimate the potential benefit of memory program optimisation before applying costly iterative compilation? While it is difficult to provide an accurate value of the expected execution time beforehand, a new technique for estimating a lower bound on execution time for scientific applications is proposed and described in this chapter.

Memory transformations and most of the current memory optimisations described in detail in chapter 3 attempt to remove cache misses. Therefore, the lower bound on execution time of a program or the potential execution time of a program after memory optimisations is defined here as the execution time of a program if all its cache misses are removed. Obtaining this lower bound on old in-order-execution processors can be relatively straightforward by using hardware counters: the execution time of the original program minus the number of misses (as recorded by hardware counters) times the memory access latency would provide an accurate lower bound on execution time [HP96]. However, modern superscalar processors described in detail in section 2.1 have non-blocking caches, out-of-order execution, complex memory hierarchies and can continue executing the program in parallel with memory accesses instead of stalling. Thus, it is not possible to deduce the no-miss execution time directly based only on the execution time of the original program and the number of misses. This is empirically demonstrated further in section 5.7 of this chapter.

Processor simulators, such as SimpleScalar described in section 3.3.2, provide a simple means to compute this lower bound, as it is trivial to modify a processor simulator so that it mimics perfect cache behaviour. However, processor simulators have several major drawbacks. Firstly, they generally model only the processor while the whole system can have a strong impact on memory performance: the way the TLB is reloaded, the bus arbitration mechanism, the physical to virtual mapping in lower cache levels and the type of memory (SDRAM, DDRAM), for example. Consequently, there is a need for a system simulator rather than a processor simulator. Secondly, it is difficult to develop a processor simulator that accurately models an existing processor without privileged access to the processor internal workings, so that an accurate system simulator would require a significant effort to accurately model the chip set, the memory chips, the operating system and all other

components. Finally, processor simulators are extremely slow: a simulated program on a current superscalar processor runs several hundred times slower than normal execution as described in section 3.3.2. Whether the simulator is used only once at the beginning of the optimisation process or worse, at each step, such a slowdown is rarely acceptable for most of the programs and is not tolerable for applications whose execution time exceeds a few minutes.

As the whole system architecture needs to be taken into account and excessive analysis time is unaffordable, simulators do not provide a satisfactory means for computing the execution time lower bound. In this chapter, a new technique that is both fast and reasonably accurate for estimating the execution time lower bound of a program is described. This technique has been implemented and tested on a wide range of programs and has been compared to other existing techniques.

5.2 Motivation and example

This section provides a motivating example, illustrating the assembler modification technique to remove almost all cache misses without affecting the remainder of the program. The general approach is to modify the program so that it retains the characteristics of the original program but induces the minimal number of misses. Therefore, the execution time of the instrumented program or its specific parts will provide a lower bound on execution time of the original program once all cache misses have been eliminated.

In numerical scientific programs where loops dominate the execution time, almost all cache misses are due to array references within these loops. The baseline of the new technique is to transform each individual array reference into a scalar reference. The memory footprint, i.e. the number of unique memory references of the resulting program is negligible compared to the original footprint and the number of misses is close to zero. The challenge is to make sure that this transformation will not affect the rest of the program and its execution on a superscalar processor.

Consider the array reference ***A[i]*** in the fortran loop in figure 5.1 (a). After compiling on the Alpha platform, this reference is translated into the assembly code shown in figure 5.1 (b), where integer register ***$19*** contains the current target address of the load instruction, i.e., the base address of array ***A*** plus loop counter ***i*** times the

```
DO i = 1, N
  ... = A[i]
  ... = B[i+1]
  ...
ENDDO
```

original code

(a)

```
lda  $19,    8($19)
ldt $f13,     ($19)  ; ... = A[i]
ldt $f14, 8192($19)  ; ... = B[i+1]
```

assembler code

(b)

```
lda  $19, 8($19)
ldt $f13,  ($28)
ldt $f14, 8($28)
```

changing memory access instructions

(c)

```
lda  $19,    0($19)
ldt $f13,     ($19)
ldt $f14, 8192($19)
```

changing address increment instruction

(d)

Figure 5.1: Assembler transformations to predict potential performance

size of one memory element (***8*** bytes in this example). ***Lda*** is a misleading acronym, it is not a load instruction but an add instruction dedicated to address computations. So in this case, it increments register ***$19*** by ***8*** to fetch the next element of array ***A***. The load instruction, ***ldt***, fetches the data located at the address stored in register ***$19*** into a floating-point register ***$f13***. These two instructions correspond to the array reference ***A[i]***.

Assume now that ***ldt*** instruction is modified as shown in figure 5.1 (c), where register ***$19*** is substituted with the constant register ***$28*** to access memory. Before executing the loop, register ***$28*** is set to a constant address which points to a memory address with preloaded data values that remain invariant throughout execution. The following ***ldt*** instructions within the loop will also use register ***$28*** to access memory but with different offsets ***8, 16*** etc. that are multiples of a single word, to point to

their own constant memory locations. Thus, the assembler instruction ***ldt $f14, 8192($20)*** corresponding to the access to the array B will be further transformed to ***ldt $f14, 8($28)***.

The new transformed code has all the same instructions; the same number of computations is performed and data dependencies are preserved between instructions operating on registers, but now addresses referenced by each instruction ***ldt*** are constant over the whole loop execution. Consequently, the memory footprint of reference ***A[i]*** is reduced from ***N x 8*** bytes to just ***8*** bytes. Considering the minimum cache size is around ***8*** Kbytes, and that the number of references is significantly less than a ***1000*** within do-loops, the memory footprint after transformation will almost always fit in cache and then only induce as many compulsory misses as the number of array references in a loop, which is negligible.

Another way of transforming the assembler code to remove cache misses also exists. If the increment of the address register ***$19*** is set to zero as shown in figure 5.1(d), then throughout the loop iterations the ***ldt*** instruction will load floating point register ***$f13*** with the same data referenced by the base address of the array ***A***. This technique gives the same performance prediction, as the first one. However, it requires a complex analysis of all instructions dealing with index calculations, and of dependencies between them and the memory access instructions. Moreover, it is platform and compiler dependent. For this reason, it has been abandoned in this research in favour of the first technique, which can be used with any language and can be easily ported to different platforms.

Naturally, once the code has been transformed as above, it no longer executes correctly. Therefore, a copy is made of each program segment of interest at the assembler level and modified as described above. First, the instrumented segment is executed and then the original segment is executed to enable normal program execution. However, the instrumented segment can still modify variables so that the program may not run correctly afterwards. For this reason, ***backup*** and ***restore*** procedures for saving and restoring all modified registers are added before and after the instrumented segment respectively. For example, consider a subroutine ***calc2*** from the SPEC benchmark swim and the transformed assembler code as shown in figure 5.2 where ***calc2_prep_*** is the backup procedure, ***calc2_tr_*** is the instrumented

```
...
br calc2_prep_     # Preparing data for transformed
                   # segment, and saving all registers

br calc2_tr_       # Executing transformed segment
br calc2_restore_ # Restoring registers
calc2_:  ...       # Executing original segment
...
```

Figure 5.2: Program modifications to ensure correct code execution after performance prediction transformation

segment, ***calc2_restore_*** is the restore procedure, ***calc2_*** is the original segment and ***br*** is the assembler instruction for branch and return. ***calc2_prep*** copies a minimal set of the data values accessed by the original segment into a new data area to be used by the modified program segment. In addition, all register values are saved and later restored. The transformed routine ***calc2_tr_*** is modified to refer to a greatly reduced number of data values residing in a special data area so that the number of cache misses is close to zero. Once it is executed, the registers are restored to their earlier values in ***calc2_restore_***. Finally, the original segment ***calc2_*** is executed. After profiling the modified program on subroutine level with high precision using hardware counters, the execution time of the transformed segment ***calc2_tr_*** will be the lower bound execution time of the original segment ***calc2_***.

In the next section, a transforming algorithm for predicting performance is described in detail. Its implementation on two platforms is also presented and is evaluated on a wide range of programs.

5.3 Performance prediction algorithm

Figure 5.3 outlines the algorithm used to determine the lower bound of the execution time. During the first step, the original program is profiled to select sections of this program that dominate the execution time, typically loop nests. During the second step, the program is instrumented and calls after each memory reference are inserted to record data values referred to by the first execution of each load/store instruction. During step three, the modified program is executed to collect and store all necessary data values. Step four is the main modification procedure. A

1. *Profile original program and select the segments of interest*
2. *Instrument program segments to collect run-time data values and addresses*
3. *Run instrumented program*
4. *Transform program:*
    - *create copies of each segment*
    - *allocate memory for preset values*
    - *transform instructions with memory access inside each segment so that they reference to preset values, analysing and keeping data dependencies*
5. *Profile transformed program*

Figure 5.3: Performance prediction algorithm

duplicate copy of the appropriate routine is created. This copy is transformed so that all array references become scalar references, and the number of memory accesses is reduced to the smallest possible footprint whilst maintaining dependences and referring to valid data. Routines for saving and restoring registers are then inserted into the program. Finally, the entire program is executed and the necessary profile data is collected.

5.3.1 Collecting data values

The purpose of the technique is to minimise references to memory in order to determine a lower bound on execution time. A naive approach would be to simply replace all load/store operations with NOOPs. However, this would alter the scheduling of the program and more importantly cause a large number of exceptions due to arithmetic on non-initialised register values. Alternatively, all load and store operations could refer to one initialised memory location, which would be permanently in L1 cache after the first reference. Although reducing floating-point exceptions, this will make every memory operation dependent on each other, radically changing the behaviour of the program. Therefore, an approach proposed here is to run the original program, obtain the values of the data referred to by each memory operation on its first execution, and then to transform all those operations to always refer to these constant values. This dramatically reduces the footprint of the program since an array reference traversing ***N*** elements of an array will now only

```
instruction_no:   load/store dest_reg, Mem[address]
                  push instruction_no
                  push address
                  br collect
```

code modification to collect data values

(a)

```
if check[instruction_no] == 0 then
    check[instruction_no] = 1
    addr[next].ins_no = instruction_no
    addr[next].add = address
    value[next] = Mem[address]
    next = next + 1
else
    Mem[instruction_no+word_size] = NOOP // overwrite 1st push
    Mem[instruction_no+2*word_size] = NOOP // overwrite 2nd push
    Mem[instruction_no+3*word_size] = NOOP // overwrite branch
endif
```

data collection procedure with self-modifying code

(b)

Figure 5.4: Data collection for performance prediction transformation

refer to the first element. This also reduces the likelihood of introduced exceptions as all memory operations reference to their own locations with the appropriately initialised values.

Obtaining the required data is achieved by inserting a jump to a data collection subroutine after each memory operation. Before jumping to the collection routine, the instruction number is pushed onto the stack, together with the memory location referred to, as shown in figure 5.4 (a), where ***instruction_no*** is simply the location in memory of the particular load/store instruction. Within the collection subroutine, the memory location and its value referred to in the original memory operation are saved to two arrays. ***Addr*** contains the ***instruction_no*** of the memory instruction plus the memory ***address*** referred to while ***value*** contains the actual value referred to i.e. ***Mem[address]***. Only the first data value referred to by an instruction is stored and therefore an additional check array is used.

*for each selected program segment (**seg**):*

- *count number of instructions with memory access (**instr_num**)*
- *allocate memory with address **addr_preset[seg]***
  *and size **instr_num * word_size** to keep preset data*

*for each instruction with memory access (**instr**) within the program segment:*

- *transform this instruction so that it references preset data*
  *with address **addr_preset[seg] + instr * word_size***

Figure 5.5: Performance prediction transformation algorithm for removing cache misses

The data collection routine that obtains all the necessary data is shown in figure 5.4 (b). Its major drawback is that the additional overhead of jumping to a subroutine on every memory access is prohibitively expensive. In some cases, it increased the execution time by a factor of 20, which can be unacceptable for large applications. To overcome this problem, self-modifying code is introduced. This code overwrites the original push and subroutine jump instructions with NOOPs (no operation instructions) once data has been collected for the first execution of any instruction. Thus, instead of jumping to the collection routine each time a load/store is executed, it only takes place once, increasing the execution time of the instrumented program for obtaining runtime data no more than 10% from the original execution time in all experiments.

5.3.2 Removing cache misses

The new technique maps all array references into scalar ones, reducing the memory footprint and the number of misses. The algorithm for this transformation is presented in figure 5.5. First, the number of instructions with memory access is counted (***instr_num***) in the assembler code for each selected program segments with the number *seg.* Then, memory is allocated with the address ***addr_preset[seg]*** and the size ***instr_num * word_size*** to keep preset data values for the transformed program to ensure correct code execution. Further, each instruction with memory access within the selected program segment is transformed to reference preset data with address ***addr_preset[seg] + instr * word_size***. The transformed code has the same instructions and the same number of calculations is performed. However, all

references within the selected program segment are constant during program execution so that the memory footprint of all references is considerably reduced in comparison with the original program.

5.3.3 Preserving data dependences

Obtaining a realistic lower-bound execution time requires preserving the properties of the original program in the transformed one. The algorithm for the performance prediction transformation, shown in figure 5.5 preserves the number and the type of all instructions, and the data dependencies between instructions operating on registers in the new code. However, it also removes all data dependencies between instructions with memory access since they refer to different locations in the specially allocated memory for preset data values.

In order to maintain the same data dependence structure of the original program, it should be ensured that if two memory access instructions reference the same memory location in the original code, they should reference the same memory location with preset data values in the transformed code. In case of dynamically allocated memory, addresses are not available at a compile time. However, the data collection procedure described in section 5.3.1 obtains run-time addresses and data values for all instructions with memory access. Comparing these run-time addresses allows one to detect instructions referencing the same memory location. Therefore, they can be further modified to reference the same memory location during performance prediction transformation.

The proposed technique for preserving data dependencies has two potential drawbacks. First, it cannot track and preserve dynamic dependencies, i.e. those data dependencies that are changing during the program execution. Second, it cannot preserve inter-iteration dependencies between instructions with memory access, i.e. it preserves data dependencies only for the first iteration of the loop. A partial solution to these problems is to obtain the lower bound of the execution time twice, with and without preserved dependencies. If the execution times are similar, then there is no influence of data dependencies on the particular program performance and, therefore, lower bound execution time is valid. If there is a considerable difference, then the obtained execution time is not guaranteed to be the lower bound. However, in all

for each selected program segment:

- *duplicate this segment to be instrumented*
  during performance prediction transformation
- *embed calls to the following procedures*
  before the selected program segment:
  - *procedure for saving the state of all registers and initialising the memory with preset data to be used by the transformed segment*
  - *procedure with the transformed program segment*
  - *procedure for restoring the state of all registers*

Figure 5.6: Algorithm to ensure correct execution of the transformed code

experiments presented in this chapter, the difference in lower bound execution times with and without preserved data dependencies between instructions with memory access is less than 1%. This can be explained by the fact that calculations are performed on the same registers in both the original and transformed programs. Therefore, data dependencies are preserved between instructions even if dynamic addresses are different in the transformed program.

5.3.4 Ensuring correct code execution

Once all array references of the selected segments of the analysed code are transformed into scalar references, the program does not execute correctly and may even crash due to the use of undefined array values by its unchanged segments. Therefore, a copy of each program segment of interest is created at the assembler level and modified as described above. First, the instrumented segment is executed and then the original segment is executed to enable normal program execution. The instrumented segment does not modify program variables as it access only specially allocated memory with preloaded data, but it still modifies registers so that the program does not yet run correctly. For this reason, *backup* and *restore* procedures for saving and restoring all registers are added before and after the instrumented segment respectively. Figure 5.6 presents an algorithm that ensures correct code execution. Three procedure calls are embedded before each selected segment. The

first procedure saves the state of all registers and initialises the memory with preset data. This data is used in the second procedure that is the copy of the original program segment transformed for the performance prediction. The third procedure restores the state of all registers.

5.3.5 Array indirection and control flow

There is a potential problem when applying the performance prediction transformation to the programs with array indirections or arbitrary control-flow. Array indirections frequently cause problems for static analysis due to compile-time unpredictability. However, since the values for all indirections are gathered during step 2, these values are saved and referred to later in the modified form of the program. Hence, indirection or other complex addressing such as tree structures do not cause difficulties. On the other hand, arbitrary control-flow does cause problems. A conditional within a loop whose value is dependent on an array element will be assigned to either true of false for the entire duration of the loop in the proposed approach. This is due to the first referenced value of the array being loaded each time for the entire loop. Currently, such references are left unmodified. Alternatively, the number of times a particular branch is taken may be recorded and replicated in the modified code to give a more accurate prediction. Such cases are the subject of the future research and are currently avoided.

5.4 Implementation

The performance prediction technique described above should be implemented in the code generation phase of a compiler in the ideal case. However, due to the inevitable lack of access to the internals of the processor vendors' compilers, this technique is implemented as a post code generation, standalone assembler modification transformation independent of high-level language. To show the portability across platforms with different instruction sets a complete toolset for the automatic analysis and instrumentation of codes has been developed for two platforms briefly described in section 4.2.2: Compaq Alpha and Intel Pentium. These are both superscalar processors with out-of-order execution and have two levels of

cache. However, the instruction sets of these processors are very different in structure and are based on RISC and CISC design philosophies, respectively. These designs are briefly described in section 2.1.1.

5.4.1 Alpha platform

Implementation of the performance transformation algorithm at assembler level requires changing instructions with memory access. The Alpha platform has a reduced instruction set where only load and store instructions can access memory and all other instructions operate on registers. Load and store instructions in the Alpha assembler have the following format:

instruction_type $data_register, offset($address_register)

Instruction_type is the type of a load or store instruction such as 'lds' for loading long word, 'ldt' for loading quad word, 'sts' for storing long word and 'stt' for storing quad word, for example. ***$Data_register*** is any floating-point register within a range of ***$f0*** .. ***$f31***. ***$Address_register*** is any integer register with memory address within a range of ***$0*** .. ***$30*** (register ***$31*** always contains the value 0).

For performance prediction, the above instructions should be changed to reference preset data in the specially allocated memory. Compaq compilers leave register ***$28*** free for other purposes. Therefore, this register is used to keep the base address of the memory with the preset data. It is initialised before executing the transformed program segment where all the instructions with memory access have their ***$address_register*** and ***offset*** replaced by the register ***$28*** and by the appropriate offset as described in detail in section 5.3.1.

5.4.2 Pentium platform

Transforming the assembler code is different on the Pentium platform, as it uses a complex instruction set. References to memory can be embedded within most of the instructions in this instruction set, unlike the reduced instruction set where only load and store instructions can access memory. The memory referencing part of instructions with memory access on the Pentium assembler has the following format:

word_type PTR immediate_address + offset1[address_register_expression]

where '***PTR***' indicates that the instruction has a memory access; ***word_type*** is the type of the used word such as '***DWORD***' for loading or storing double words or '***QWORD***' for loading or storing quad words, for example. The address part of the instruction may consist of an ***immediate address*** and its ***offset*** plus an ***address_register_expression*** that can be a complex linear expression such as ***register1+register2*const***.

To predict performance, the address part of the instructions should be changed to reference memory with preset data. Since immediate addresses are allowed in instructions on the Pentium platform, the address part is simply replaced to give the following expression:

*word_type PTR addr_preset[seg] + instr * word_size*

where ***addr_preset[seg]*** is the immediate address of the memory with preset data and ***instr * word_size*** is the offset of the preset data for the particular instruction, described in detail in section 5.3.1. The techniques developed for the Pentium and Alpha platforms demonstrate that little platform specific modifications are required even for radically distinct ISAs.

5.5 Experimental results

The experiments for determining a lower bound on execution time are performed on both the Alpha and Pentium. The same programs are selected as in the previous chapter: matmul, sor and eight benchmarks from the SPEC'95 benchmark suite using reference datasets. The most time-consuming loops of these programs are selected for the performance prediction transformation. Due to architectural differences of the Alpha and Pentium platforms, the program execution also varies on these platforms. Therefore, the most time-consuming loops are not necessarily the same on both platforms. These program segments are further transformed to obtain a lower bound on execution time. The execution times of the original and transformed versions and their respective IPC (instruction per cycle) are measured using a high precision profiler and hardware counter support. These figures and the potential speed-up for each program segment are presented in table 5.1 for the Alpha platform and in table 5.2 for the Pentium platform.

Program:	Procedure and loop number:	Original time:	Original IPC:	Transf. time	Transf. IPC:	Speedup:
matmul	main_1	31.1	0.3	3.3	2.5	9.4
sor	main_1	48.4	0.5	28.6	0.8	1.7
tomcatv	main_1	28.9	1.0	12.3	2.4	2.3
	main_2	8.4	0.5	4.8	1.0	1.8
	main_3	19.3	0.3	4.8	1.6	4.0
	main_4	10.0	0.8	2.4	3.3	4.2
	main_5	11.5	0.6	2.4	2.2	4.8
swim	calc1_1	19.9	1.0	9.3	2.3	2.1
	calc2_1	25.0	1.1	9.4	2.9	2.7
	calc3_1	24.0	0.9	6.5	3.2	3.7
su2cor	adjmat_1	3.9	1.4	1.6	3.4	2.4
	bespol_1	3.6	2.4	2.6	3.5	1.4
	matadj_1	4.0	1.4	1.7	3.4	2.4
	matmat_1	10.8	1.2	4.0	3.4	2.7
	sweep_2	3.5	0.6	0.7	3.1	5.0
mgrid	Psinv_1	22.0	1.9	18.6	2.3	1.2
	resid_1	43.4	1.9	34.9	2.3	1.2
	rprj3_1	7.4	1.0	3.7	1.9	2.0
applu	buts_1	16.0	0.7	6.4	1.7	2.5
	jacu_1	12.9	0.9	5.2	2.3	2.5
	rhs_3	3.9	1.5	2.4	2.4	1.6
	rhs_4	4.1	1.5	2.6	2.3	1.6
turb3d	dfct_1	19.6	0.8	5.9	2.6	3.3
	dfct_2	11.0	2.0	6.6	3.4	1.7
	Trans_1	8.1	2.6	7.8	2.7	1.0
apsi	hyd_1	4.2	0.5	1.3	1.5	3.2
	Leapfr_2	3.2	0.5	0.7	2.5	4.6
	radb4_1	1.0	3.3	1.0	3.3	1.0
	radb4_2	3.3	2.5	3.3	2.5	1.0
	radf4_2	2.1	2.5	2.1	2.5	1.0
	trid_1	4.0	0.6	2.8	0.9	1.4
	trid_2	3.8	0.5	2.3	0.9	1.7
	ucrank_1	2.0	2.0	1.9	2.0	1.0
wave5	parmvr_1	1.4	1.0	1.1	1.3	1.3
	parmvr_3	8.2	0.8	3.7	1.8	2.2
	parmvr_4	4.9	2.1	2.6	3.0	1.9
	parmvr_5	1.1	1.3	1.1	1.3	1.0
	parmvr_11	4.5	1.0	1.8	2.6	2.5

Table 5.1: Original and lower-bound execution times with IPCs (Alpha platform)

The results obtained demonstrate large variations in potential performance improvement among various program segments of the examined applications. Most of the programs on both platforms contain both loops with a very high potential speed-up and those with negligible potential. For example, loop main_1 from the matmul kernel has a high potential speed-up of 9.4 on the Alpha platform. On the same platform, loops main_5 from the tomcatv benchmark, sweep_2 from the su2cor

Program:	Procedure and loop number:	Original time:	Original IPC:	Transf. time	Transf. IPC:	Speedup:
matmul	main_1	83.9	0.1	6.2	1.6	13.5
sor	main_1	48.9	0.3	13.0	1.2	3.8
tomcatv	main_1	47.4	0.4	21.5	1.2	2.2
	main_2	13.4	0.4	3.7	1.4	3.6
	main_3	32.5	0.1	11.3	0.5	2.9
	main_4	25.6	0.1	3.2	1.6	8.0
	main_5	24.3	0.1	1.8	1.4	13.5
swim	calc1_1	41.7	0.4	25.3	0.7	1.6
	calc2_1	40.7	0.3	13.7	1.1	3.0
	calc3_1	48.8	0.2	9.3	1.4	5.2
su2cor	adjmat_1	14.7	0.3	3.5	1.2	4.2
	bespol_1	15.8	0.2	2.9	1.1	5.4
	matadj_1	15.7	0.3	3.5	1.1	4.5
	matmat_1	37.7	0.3	8.5	1.2	4.4
	sweep_2	9.8	0.1	0.7	1.4	14.0
	sweep_3	2.0	0.1	1.7	0.1	1.2
	sweep_4	3.1	0.5	2.8	0.7	1.1
mgrid	psinv_1	49.8	0.4	26.3	0.9	1.9
	resid_1	105.0	0.4	47.7	1.0	2.2
	rprj3_1	11.8	0.2	3.3	1.0	3.6
applu	buts_1	34.7	0.5	15.5	1.2	2.2
	jacu_1	29.3	0.3	10.3	1.0	2.8
	rhs_2	6.7	0.5	4.5	0.9	1.5
	rhs_3	6.9	0.5	4.5	0.9	1.5
	rhs_4	7.2	0.5	4.4	0.9	1.6
turb3d	dfct_1	31.7	0.2	7.6	1.6	4.2
	dfct_2	10.5	0.9	7.7	2.0	1.4
	fftz2_1	46.6	1.6	45.1	1.6	1.0
	trans_1	8.7	1.2	8.7	1.2	1.0
apsi	dtdtz_1	4.6	0.1	1.0	0.6	4.6
	dvdtz_1	4.9	0.2	1.1	0.6	4.5
	leapfr_2	5.0	0.1	0.5	1.2	10.0
	trid_1	5.2	0.3	4.9	0.3	1.1
	trid_2	4.9	0.1	4.5	0.1	1.1
	ucrank_1	3.8	0.7	3.6	0.8	1.1
wave5	parmvr_1	4.1	0.1	4.1	0.1	1.0
	parmvr_3	14.7	0.2	3.0	1.2	4.9
	parmvr_4	15.3	0.3	5.2	0.9	2.9
	parmvr_5	4.5	0.1	4.5	0.1	1.0
	parmvr_11	8.1	0.3	2.0	1.6	4.1

Table 5.2: Original and lower-bound execution times with IPCs (Pentium platform)

benchmark and leapfr_2 from the apsi benchmark have a potential speed-up close to 5. On the other hand, loops psinv_1 and resid_1 from the mgrid benchmark, trans_1 from the turb3d benchmark, radb4_1, radb4_2, radf4_2 and ucrank_1 from the apsi benchmark and parmvr_5 from the wave5 benchmark have a negligible potential performance improvement.

The potential speed-up can be used to drive further optimisations. The performance prediction technique can provide information about whether a loop is memory bound, i.e. when data is retrieved from memory slower than it can be processed, or not. In this way, it is similar to the static optimisation technique proposed by Carr and Kennedy and described in detail in section 3.2.1. However, the new performance prediction technique proposed in this chapter considers all the hardware and program run-time parameters and is precise.

When a loop has a high potential speed-up, it is memory bound and can hence benefit from further memory optimisations. If the potential speed-up is negligible, the loop is either balanced or is compute bound, i.e. the rate of data retrieval from memory is faster than its processing rate. In this case, this loop will not benefit from memory optimisations. This can help to reduce the search space for iterative compilation by removing those loops from consideration that do not have potential speed-up as described in the next chapter. Compute-bound loops can benefit from optimisations that improve ILP such as software pipelining and loop unrolling, for example. However, ILP optimisations are beyond the scope of this research.

Experiments performed on the Pentium platform show that loops with high potential speed-up are similar on both Alpha and Pentium platforms. This is explained by the fact that the execution time of memory bound loops where memory accesses dominate depends primarily on the memory design that is similar on both architectures. For example, loops main_1 from the matmul kernel, main_5 from the tomcatv benchmark, sweep_2 from the su2cor benchmark and leapfr_2 from the apsi benchmark have high potential speed-ups as on the Alpha platform, though to a different extent due to differences in the memory bandwidth of these architectures. The potential speed-up of compute-bound loops should also be negligible on both platforms as it does not depend on the memory access time and cache misses. Thus, loops trans_1 from the turb3d benchmark, ucrank_1 from the apsi benchmark and parmvr_5 from the wave5 benchmark have a potential speed-up close to 1 on both platforms and are compute bound. All other loops have different potential speed-ups on both architectures as they heavily depend on both the processor architecture and the memory design. For example, loops psinv_1 and resid_1 from the mgrid benchmark have relatively high potential speed-ups of 1.9 and 2.2 respectively on the

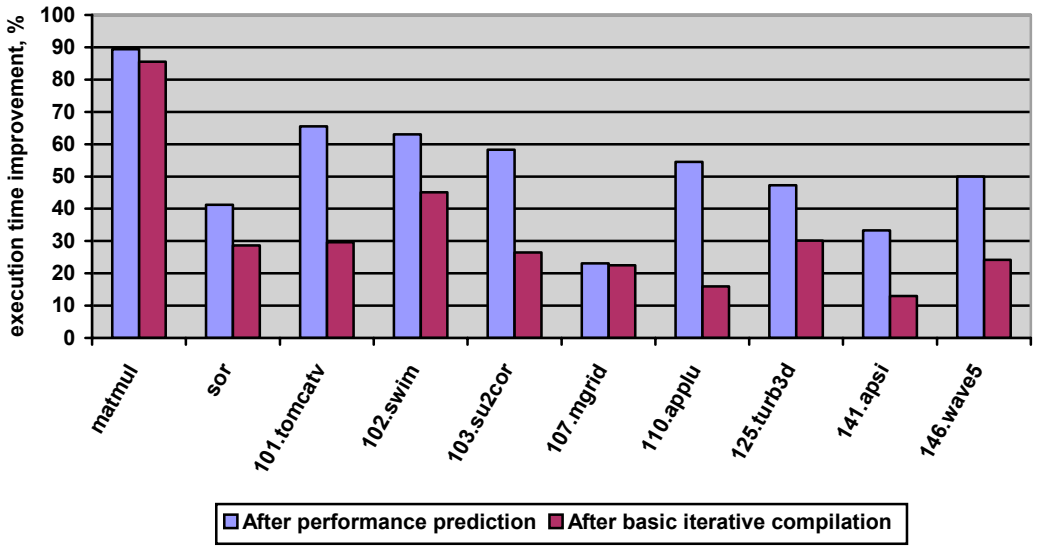
Pentium platform while these loops have a negligible potential speed-up on the Alpha platform.

It should also be noted that the IPC of the transformed loops varies considerably on both architectures and is not the ideal one (which is 4 for Alpha and 3 for Pentium). Therefore, it is not possible to obtain the lower bound execution time simply by multiplying the number of executed instructions and the ideal IPC for the targeted machine.

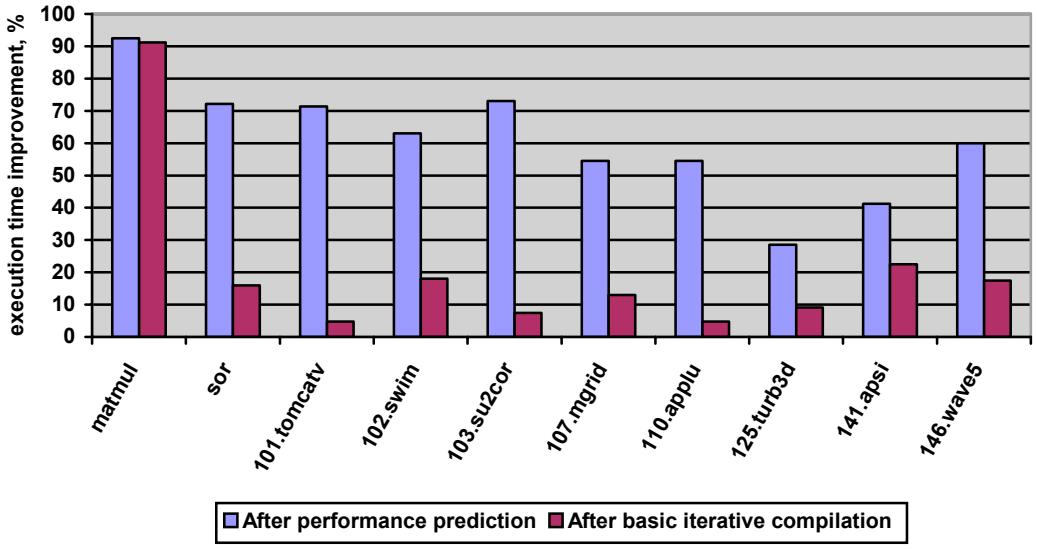
Figure 5.7 shows the overall potential performance improvement and the execution time improvement after iterative compilation with the basic search strategy described in chapter 4 for each program on both processors. It demonstrates that although it is not guaranteed that the lower bound of the execution time can be achieved through selected transformations, performance improvements of some programs are close to the predicted potential improvements after iterative compilation with the basic search strategy. Besides, it shows that iterative compilation even with only three memory transformations is an efficient optimisation technique. For example, sor on the Alpha platform and matmul on both platforms achieve considerable performance improvements close to the potential ones. Thus, the lower bound execution time can be used as a realistic criterion to drive and stop optimisation process.

Though it is possible to achieve the potential performance improvement for small kernels with simple loop structures due to easy, straightforward and efficient memory optimisations of such code, it is not the case for larger complex applications. For the Alpha platform, performance improvements of only three SPEC benchmarks - swim, mgrid and turb3d are relatively close to the potential ones. The performance improvements of all other benchmarks though considerable are still far from the best ones. This can be explained by the limited number of transformations used for iterative compilation. For example, these transformations do not tackle compulsory misses and memory bandwidth problem that can be efficiently optimised using prefetching as described in section 3.1.5 but is out of the scope of the thesis.

Differences between potential performance improvement and improvement after iterative compilation are even more dramatic on the Pentium platform. There are big gaps in potential and achieved performance improvement for all SPEC benchmarks.



(a) Alpha platform



(b) Pentium platform

Figure 5.7: Overall potential and iterative performance improvement (%)

As in the case of the Alpha platform it can partly be explained by the complex loop structures of those programs and by lower efficiency of the selected transformations but more importantly are the differences in architectures of those two platforms. The Pentium platform has a slower memory system than the Alpha platform and therefore the potential for the improvement is higher when all cache misses are removed. However, CISC architectures have inherently higher instruction and memory latencies than RISC architectures. Therefore, the outcome of optimisations is smaller on the Pentium platform than on the Alpha platform. Nevertheless, it does not mean

that the lower bound cannot be achieved as matmul achieved the potential performance improvement after iterative compilation on the Pentium platform, for example. To achieve this performance other optimisations should be considered and the influence of various transformations on each other should be analysed, which is the topic of future research. Therefore, performance prediction technique can also be used to analyse the efficiency of various optimisation techniques and transformations for the given programs and architectures.

5.6 Performance validation

To fully validate the fact that the instrumentation only affects memory behaviour and that the lower bound can effectively be interpreted, the following experiment is performed using a full processor simulator. The Alpha 21264 processor is modelled using the SimpleScalar tool described in section 3.3.2. This model is also modified in such a way that its cache is perfect, i.e., all memory requests hit in the first-level cache. Further, both the original swim program from the SPEC'95 benchmark and its transformed version for the performance prediction are executed on this simulator with normal and perfect cache. Since the performance prediction transformation removes all cache misses, the performance of the transformed program on the simulator with normal cache should be nearly identical to the performance of the original program on the simulator with perfect cache. Results presented in table 5.3 confirm that instrumentation barely affects the overall program behaviour. The IPC of the transformed program run on the simulator with either normal or perfect cache, 3.02, is near to the one of the original program when simulated with perfect cache, which is equal to 2.98. Results presented in table 5.4 show the number of cache accesses and misses for the original program and its transformed version run on the simulator with normal cache. These results also confirm that performance prediction transformation removes most of the L_1 cache misses and most of the L_2 cache accesses (large L_2 cache miss ratio for the transformed code is not important since the total number of L_2 cache accesses in the transformed program is negligible in comparison with the original program).

	Original program:	Program transformed for performance prediction:
IPC (simulator with normal cache):	2.42	3.02
IPC (simulator with perfect cache):	2.98	3.02

Table 5.3: IPC of the original and transformed programs obtained using the simulator with normal and perfect caches

	Original program:	Program transformed for performance prediction:
Number of L_1 cache accesses:	295,705,805	298,213,871
Number of L_1 cache misses:	7.2%	0.0%
Number of L_2 cache accesses:	2,123,885	1,993
Number of L_2 cache misses:	72.4%	61.62%

Table 5.4: Cache behaviour of the original and transformed programs

5.7 Comparison with existing techniques

Many existing optimisations or performance prediction techniques, described in detail in chapter 3, attempt to predict and reduce the number of cache misses. It may be argued that information about the number of cache misses obtained through hardware counters can either guide optimisations or predict performance with less effort. Such techniques can attempt to determine the overhead due to memory access time directly using the information about cache misses obtained by hardware counters, subtract this from the original time to obtain the lower bound on execution time. These techniques may work well on old in-order-execution processors by using the following formula for CPU execution time:

*CPU execution time = (CPU clock cycles + memory stall cycles) * Clock cycle*

However, modern superscalar processors with non-blocking caches and out-of-order execution can considerably overlap CPU time and memory stall time, invalidating this formula. Furthermore, the impact of memory access can be severely underestimated by hardware counters. The following example illustrates this statement. Consider matmul shown in figure 5.8 (a). This kernel is executed on the Pentium platform and is profiled by the VTune tool [Int03b] using hardware

```
DO I=1, N
 DO J=1, N
  DO K=1, N
   A(I,J)=A(I,J)+B(I,K)*C(K,J)
  END DO
 END DO
END DO
```

original matmul

(a)

```
DO I=1, N
 DO J=1, N
  DO K=1, N
   A(I,J)=A(I,J)+B(I,K)*C(K,J)+(B(I,K)+C(K,J))*(B(I,K)-C(K,J))
  END DO
 END DO
END DO
```

synthetically generated kernel

(b)

Figure 5.8: Original matmul and synthetically generated kernel

counters. The performance prediction technique is then applied to this kernel. Finally, it is optimised using iterative compilation with the basic search strategy described in chapter 4. The execution time, the number of data memory references and the miss ratio for L_1 and L_2 caches are shown in table 5.5 (a) for the original matmul and for its transformed and optimised versions. These results show that the original kernel exhibits a high number of cache misses on both cache levels. The performance prediction technique shows how this program would behave when all cache misses are removed. Iterative compilation with the basic search strategy is capable of eliminating most of the cache misses for this kernel so that its optimal execution time is close to the predicted lower bound time as expected.

Another kernel shown in figure 5.8 (b) is synthetically generated from matmul. It is profiled on the Pentium platform using VTune tool. This kernel performs more calculations but on the same array references so that the overall number of data references and cache misses is the same. These figures are shown in table 5.5 (b), where a slight difference in the number of cache misses is determined by the hardware counter precision. Moreover, this kernel is also generated in such a way that its execution time is nearly the same as that of the original matmul, meaning that in both cases the memory accesses dominate the execution and all calculations are

	Execution time:	Number of data memory references:	L_1 cache miss ratio:	L_2 cache Miss ratio:
Original kernel:	86.2 s.	1.3E9	0.452	0.448
Transformed kernel for performance prediction:	6.2 s.	1.3E9	0.000	0.000
Optimised kernel after iterative compilation:	7.6 s.	1.3E9	0.013	0.006

original matmul

(a)

	Execution time:	Number of data memory references:	L_1 cache miss ratio:	L_2 cache Miss ratio:
Original kernel:	86.7 s.	1.3E9	0.445	0.447
Transformed kernel for performance prediction:	18.8 s.	1.3E9	0.000	0.000
Optimised kernel after iterative compilation:	19.5 s.	1.3E9	0.009	0.004

synthetically generated kernel

(b)

Table 5.5: Example demonstrating the advantage of the proposed performance prediction technique over the existing ones that are based on counting the number of cache misses

performed in parallel with memory stalls. In this case, optimisation techniques based on hardware counters, would expect the resulting optimised code for the modified matmul to have the same execution time as the optimised version of the original matmul. However, the performance prediction technique gives a potential lower bound on execution time for the new kernel approximately three times higher then that of the original kernel. This is validated by iterative compilation - 19.5 s. vs 7.6 s.

Hardware counters techniques may also attempt to predict the lower bound on execution time using the following formula for the in-order processor [HP96]:

$$\textit{Memory access overhead} = \textit{Data references} * (HitRate_{L1} * HitTime_{L1} + MissRate_{L1} * (HitRate_{L2} * HitTime_{L2} + MissRate_{L2} * HitTime_{Main\ Memory}))$$

For the above example, there are $1.3*10^9$ data references and the average hit times is measured as follows: $HitTime_{L1} = 1.5\ ns$; $HitTime_{L2} = 8\ ns$; $HitTime_{main\ Memory} = 152\ ns$. Note that these figures depend on the processor and system configuration. After

substituting the values from table 5.4 for the original matmul, the memory overhead is:

$$\textit{Memory access overhead} = 1.3*10^{9} * ((1-0.452)*1.5 + 0.452*((1-0.448)*8 + 0.448*152))*10^{-9} = 43.7\ s.$$

Using the simplified equation above leads to CPU computation time of $86.2 - 43.7 = 42.5\ s.$ If all cache misses are removed so that all accesses are to L_1 cache only, the memory stall obtained from the above formula is 2 s. Thus the lower bound calculated solely from hardware counters is 42.7 s + 2 s = 44.7 s. However, this is 5.9 times higher than the time of the highly tuned matmul and its lower bound predicted by technique presented in this chapter (44.7 s. versus 7.6 s.) and is 2.3 times higher than the lower bound of the synthetically generated kernel (44.7 s. versus 19.5 s). Therefore, the performance prediction technique described in this chapter provides a more realistic lower bound on the execution time. Furthermore, iterative compilation with the basic search strategy is capable of eliminating most of the cache misses for the modified matmul as well as for the original matmul so that their execution time are close to the predicted ones. This result demonstrates the advantage of using the performance prediction technique to obtain the lower bound execution times. Although, the lower bound execution time for the last kernel could be predicted using simulation, it will be thousands of times slower than the proposed performance prediction technique. Furthermore, the new method is superior to current techniques by being able to predict a lower bound on execution time for a particular application on a target platform without architectural knowledge of this platform and with a minimal amount of knowledge about the instruction set of this platform.

5.8 Summary

This chapter describes a new technique for the fast evaluation of the lower bound on execution time of program segments assuming that most cache misses have been removed. It is based on assembler modification and is accurate as all instructions are the same in the transformed code but load and store instructions refer to constant addresses. Data dependencies are preserved and the program is actually run on the targeted machine thus taking into account all system and architecture parameters. The performance prediction technique is validated on a cycle accurate simulator.

However, it is significantly faster than simulation since the execution time of the instrumented program is at most twice the execution time of the original program compared with a 500 to 2000 times slowdown for simulation based techniques. It also demonstrates that a majority of existing optimisations or performance prediction techniques that attempt to predict and reduce the number of cache misses are no longer valid on modern superscalar processors with non-blocking caches and out-of-order execution.

Though this technique does not guarantee whether or not the lower bound of the execution time can be achieved through transformations, it can determine program segments which have a memory problem and which are candidates for memory optimisations. The following chapter investigates the use of this lower bound calculation in predicting the performance improvement and in reducing the transformation space for advanced iterative compilation approaches to program optimisations.

Chapter 6

Search Space Reduction

This chapter extends iterative compilation beyond the basic search strategy described in chapter 4 by using the performance prediction technique presented in chapter 5 with a random search strategy. This dramatically reduces the number of iterations needed from thousands to less than a hundred and still achieves considerable performance improvements. Thus, iterative compilation becomes a realistic optimisation approach not only for the small kernels but also for a broad range of applications. The results obtained are compared with the basic search strategy and with other existing optimisation methods. Finally, a distinct method that can reduce the iterative compilation time by using smaller datasets during program optimisation is briefly considered.

6.1 Introduction

The two previous chapters describe an iterative optimisation technique that outperforms current commercial compilers and introduce a technique for determining a lower bound on execution time of a program. However, the compilation time for the iterative search is excessively high (thousands of iterations) making it usable only when the lifetime of a program is much higher than the time spent during its optimisation. Therefore, the goal is to dramatically reduce the number of iterations without sacrificing performance.

This chapter describes a technique to significantly reduce compilation times with only 1-3% reduction in performance. It is achieved by using performance prediction to remove loops that do not have any potential speed-up from the iterative search, and by using a new random search strategy that investigates only a few random transformation factors instead of all possible ones.

	Execution time:
Original kernel:	83.9s.
Original kernel transformed for performance prediction:	6.2 s.
Optimised kernel after iterative compilation (approximately 1600 iterations):	6.8 s.
Optimised kernel transformed for performance prediction:	6.4 s.

(a) matmul

	Execution time:
Original kernel:	48.9 s.
Original kernel transformed for performance prediction:	13.0 s.
Optimised kernel after iterative compilation (approximately 1600 iterations):	41.1 s.
Optimised kernel transformed for performance prediction:	13.6 s.

(b) sor

Table 6.1: Example demonstrating the use of the performance prediction technique in iterative compilation (Pentium platform)

6.2 Using performance prediction

The performance prediction technique, described in chapter 5, determines a lower bound on execution time for arbitrary sections of a program. In practice, if used before applying a time-consuming iterative search for the best transformations, it can reduce the search space by selecting only those sections of the program that have the potential to be improved. This technique is fast. It needs only one preliminary run to collect various run-time information about a program that is only 10-15% slower than the execution time of the original program and a single run of the transformed code that is at most twice as slow as the original program. This time is negligible in comparison with the time needed to complete the iterative optimisation process, thus ensuring that there is no overall slowdown using this technique.

To demonstrate the use of the performance prediction technique, matmul is analysed on the Pentium platform and results are presented in table 6.1 (a). The original time of this kernel is **83.9** s. and the predicted lower bound is **6.2** s. This means that there is a great potential for this kernel to be improved and therefore it should be further optimised. Iterative compilation is capable of improving the performance of the original kernel considerably by **92.6%** after approximately 1600

iterations. The execution time of the optimised kernel is **6.8** s. that is close to the lower bound. Assume now, that the analysed code is already optimised. Performance prediction technique can be used to detect such cases. For example, applying this technique to the optimised kernel after iterative compilation provides a lower bound execution time of **6.4** s, which gives a performance improvement of approximately **6%**. Therefore, there is little potential for this code to be further optimised and it can be excluded from consideration.

Although the prediction technique suggests which sections of the program have a potential to be improved, it does not guarantee that the lower bound execution time will be achieved through optimisation. To demonstrate this issue, consider the results for sor shown in table 6.1 (b). The original time of this kernel is **48.9** s. and the lower bound time is **13.0** s., which means that is **73.4**% of the potential performance improvement. However, the time achieved after iterative compilation is **41.1** s. that is about **16.0%** improvement. Such a big gap for sor on the Pentium platform is explained in section 5.5 and is briefly due to higher instruction and memory latencies on CISC architectures that demonstrate a high potential for improvement but are harder to achieve through selected optimisations than say on the Alpha platform. Nevertheless, it does not mean that the lower bound cannot be achieved. For example, matmul achieved the potential performance improvement after iterative compilation on the Pentium platform. Other optimisations such as prefetching should be considered, for example, to achieve this performance, but are beyond the scope of this thesis. Therefore, the performance prediction technique can be used for identifying those sections of the program that have a potential for improvement after memory optimisations and for excluding those loops from the iterative search that do not, thereby reducing the search space for iterative compilation.

6.3 Random search strategy

Using the performance prediction technique reduces the search space by selecting only those loops for further iterative compilation that have a potential for improvement. However, the most time consuming part of the iterative search strategy is applying all possible transformation factors within the chosen range.

1. *profile original program*
2. *run performance prediction technique*
3. *choose set of arrays and loops that dominate the execution time and have the potential for the improvement*
4. *apply data transformations:*
   - *apply array padding **N** times with a random padding factor (**1..N_a**) for all global arrays*
   - *run program variant and record the best execution time*
   - *select the best transformation (minimal execution time)*
5. *apply loop transformations:*

for each selected loop nest:

for each loop from this nest:

if loop is not innermost and is within a perfect nest:

- *apply loop tiling **N** times with a random tiling factor (**1.. N_t**) for the loop nest*
- *run program variant and record the best execution time*

if loop is innermost:

- *apply loop unrolling **N** times with a random unrolling factor (**1..N_u**) without tiling*
- *run program variant and record the best execution time*

if the best tiling factor is found for the enclosing iterators within the loop nest:

- *choose best tiling transformation*
- *apply loop unrolling **N** times with a random unrolling factor (**1..N_u**) for the innermost loop*
- *run program variant and record the best execution time*

select the best transformation for the loop nest

(either loop unrolling or a combination of both loop tiling and loop unrolling)

Figure 6.1: Random search strategy algorithm

To tackle this problem, new search algorithms should be used that apply fewer transformation factors. Kisuki et al. evaluates five search algorithms on three

benchmarks in [KKO+00]: genetic search, simulated annealing, pyramid or grid search, window search and random search. The grid or pyramid search strategy defines a top level grid over the search space from the basic strategy. Each point on this grid is ordered into a priority queue and is evaluated. The grid can be further redefined around the best points that minimise program overall execution time. The window search strategy defines windows over the search space. At first, there is only one window that covers the entire space. During iterative compilation, a number of samples is taken and ordered into a priority queue. Smaller windows are further defined around the best points and evaluated again. The random search strategy is the simplest one out of all presented here and picks transformation factors randomly during a given number of iterations.

Simulated annealing is a search algorithm for a minimum in any general system using a rough analogy with a physical process of heating and then slowly cooling a metal into a minimum energy crystalline structure. During iterative compilation, the aim of this algorithm is to minimise the execution time of a program. At first, a point is selected randomly from the search space and all its neighbouring points are inspected. The system is subsequently moved to points with lower execution time. However, from time to time it is allowed to jump to a point with higher execution time to avoid potential traps in the local minima.

Genetic search is also used to find the minimum execution time and is based on an analogy with evolution of living organisms. At first, an initial population consisting of 20 programs with random parameters is created. Second, a bit representation of tile and unroll factors is created and a crossover point is determined for a number of program variants. Further, the upper and the lower halves of this bit representation or “chromosomes” are concatenated. During the mutation phase, the remaining bits are flipped and the new population of programs is evaluated. Only 20 programs with the minimum execution time are left in the new population and the rest is deleted.

The evaluation of all these strategies shows that iterative compilation is capable of achieving high levels of optimisation in all cases. Furthermore, there is no significant difference in their efficiency – all the obtained speed-ups are within 5% of each other on average. However, there is a difference in the number of iterations to

	basic search strategy	random search strategy
Execution time of matmul:	83.9 s.	
Performance prediction time for matmul:	6.2 s.	
Execution time of the optimised matmul after iterative compilation:	6.8 s.	7.5 s.
Performance improvement after iterative compilation:	91.9%	91.1%
Number of iterations needed:	1599	20

Maximum array padding factor $N_a = 64$
Maximum loop tiling factor $N_t = 512$
Maximum loop unrolling factor $N_u = 512$

Number of random tries for the random search strategy $N = 5$

Table 6.2: Comparison of the basic and random search strategies (matmul, Pentium platform)

obtain maximum speed-ups. Grid search is the slowest to reach the minimum of the execution time as the original grid is defined over the whole search space. Simulated annealing and random search strategies are the fastest and, finally, genetic and windows search strategies are in between. The results from this paper demonstrate that only a small fraction of the original search space is needed (0.03 to 0.05%) to reach 90% of the maximum speed-up after the basic search strategy.

Due to the simplicity and efficiency of the random search strategy, a new modified algorithm is proposed here and presented in figure 6.1. It differs from the algorithm presented in chapter 4 by using the performance prediction technique and by using random factors, which is enough to reduce the number of iterations by up to two orders of magnitude. Selecting the same transformation factor is obviously wasteful and is avoided. Both algorithms are able to optimise not only small kernels but real large applications as well, unlike other iterative compilation methods described above.

To demonstrate the advantages of the random search over the basic one described in chapter 4, matmul is optimised on the Pentium platform using both these strategies. The results presented in table 6.2 show the basic search strategy achieves 92.6% performance improvement after 1599 iterations. On the other hand, applying the random search strategy achieves similar result of 91.7% improvement. However,

it needs 80 times fewer iterations to achieve this result. In other cases, performance improvement can be lower, but the number of iterations needed to achieve it varies from 20 to 80 that is two orders of magnitude less than after using the original basic search strategy.

6.4 Experimental results

The random search strategy is evaluated in a similar manner to the basic search strategy described in chapter 4. All applications are evaluated on both the Alpha and Pentium platforms. To compare results with static and dynamic optimisations of the best state-of-the-art commercial compilers the following three compiler options are used:

Opt.1) maximum internal optimisations with data and loop transformations disabled;

Opt.2) maximum internal optimisations with data and loop transformations enabled;

Opt.3) feedback-assisted optimisations.

All applications are first profiled to choose the subroutines that dominate execution time. Within each chosen subroutine, all loop nests are selected and the performance prediction technique is applied as described in chapter 5 to eliminate those loops from the search space that have a negligible execution time or have a little potential for improvement after applying memory optimisations of less than 10-15%, for example. Table 6.3 shows that there is a significant difference between the total number of loops in a program and the number of loops that have been selected for optimisation. This is due to the fact that SPEC benchmarks consist of a large number of loops, but not all of them dominate the execution time or have any potential for further improvement. Since most of the SPEC FP benchmarks are memory bound, only about 15% of loops that dominate execution time have been excluded from the search space after the performance prediction technique. All arrays referenced within the chosen loops are considered further. The maximum array padding factor, loop tiling factor and loop unrolling factors are the same as in the case of the basic search strategy: N_a = 64, N_t = 512 and N_u = 512, respectively. The difference between this search strategy and the basic search strategy described in

Application:	Total number of loops:	Number of selected loops for the random search strategy:
matmul	6	2
sor	5	2
tomcatv	16	5
swim	24	6
su2cor	117	4
mgrid	46	5
applu	168	5
turb3d	70	6
apsi	144	5
wave5	362	15

Table 6.3: Total number of analysed loops and the number of selected loops for the random search strategy

chapter 4 is in trying only ***N*** random factors for each transformation instead of checking the whole range of all possible factors. If the number of tries is too high, the overall number of iterations is close to the one of the basic iterative search and the performance improvement is close to the improvement after basic search strategy. If the number of tries is too small, the overall number of iterations is small as well. In this case, the probability of finding the best transformation factors is low and the performance improvement may be negligible. After considering the results from the paper [KKO+00] and aiming to perform less than a maximum of one hundred iterations on the optimisation process to make iterative compilation a realistic approach for a broad range of applications, the number of random tries ***N*** for each transformation factor is chosen to be 5 for all kernels and SPEC benchmarks.

Table 6.4 presents the execution time improvements achieved after applying iterative compilation with the random search strategy relative to Opt.1, Opt.2 and Opt.3. Table 6.5 and figure 6.2 compare the performance improvements after iterative compilation with the basic and random search strategies and present the

Application:	Number of iterations:	Execution time improvements:		
		Over Opt.1	Over Opt.2	Over Opt.3
matmul	20	79.2%	69.8%	69.8%
sor	20	11.2%	11.2%	11.2%
average (kernels)	20	45.2%	40.5%	40.5%
tomcatv	80	20.2%	23.3%	22.2%
swim	50	43.3%	33.6%	33.6%
su2cor	50	24.8%	14.1%	14.7%
mgrid	40	15.5%	14.8%	10.0%
applu	55	15.1%	4.4%	5.9%
turb3d	45	27.6%	39.3%	33.9%
apsi	65	11.5%	8.7%	9.0%
wave5	80	23.4%	13.7%	9.9%
average (benchmarks)	58	22.7%	19.0%	17.4%

(a) Alpha platform

Application:	Number of iterations:	Execution time improvements:		
		Over Opt.1	Over Opt.2	Over Opt.3
matmul	20	91.7%	91.5%	91.6%
sor	20	15.6%	15.6%	15.6%
average (kernels)	20	53.7%	53.6%	53.6%
tomcatv	80	2.4%	2.1%	-0.5%
swim	50	17.9%	18.4%	17.9%
su2cor	50	6.5%	6.2%	5.7%
mgrid	40	12.9%	12.9%	13.8%
applu	55	0%	-2.1%	-3.5%
turb3d	45	8.8%	8.0%	8.2%
apsi	65	22.2%	21.7%	23.9%
wave5	80	17.3%	16.0%	15.7%
average (benchmarks)	58	11.0%	10.4%	10.2%

(b) Pentium platform

Table 6.4: Execution time improvements (%) after iterative compilation with the random search strategy over Opt.1, Opt.2 and Opt.3

number of iterations needed to achieve these performance improvements for both strategies.

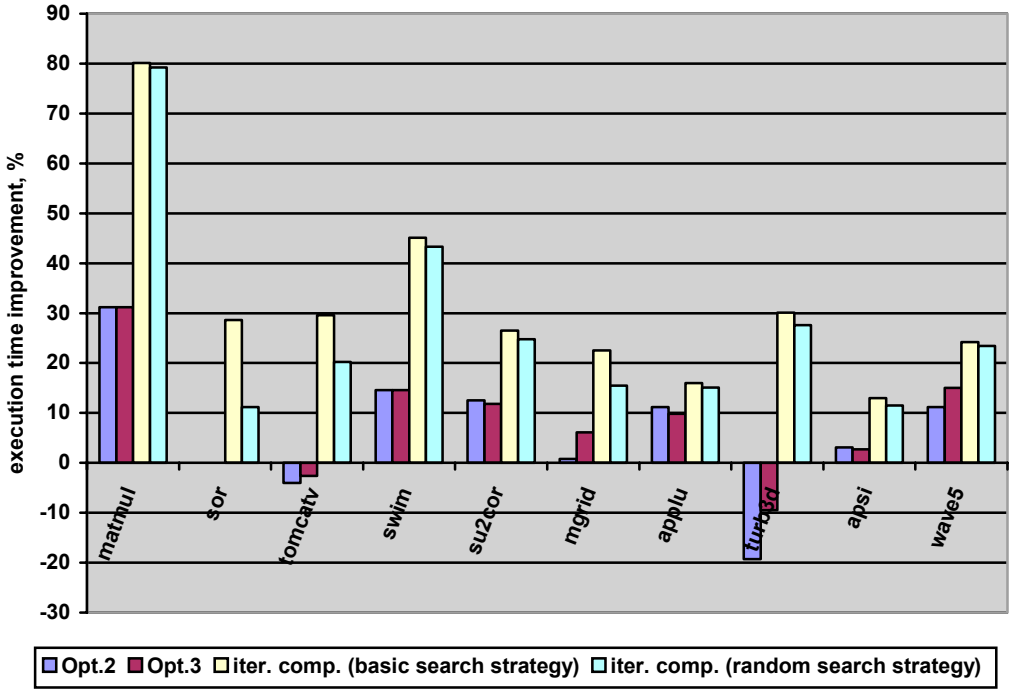
	Random search strategy		Basic search strategy	
	Performance improvement:	Number of iterations:	Performance improvement:	Number of iterations:
matmul	79.2%	20	80.1%	1599
sor	11.2%	20	28.6%	1599
average (kernels)	45.2%	20	54.4%	1599
tomcatv	20.2%	80	29.6%	7738
swim	43.3%	50	45.1%	6205
su2cor	24.8%	50	26.5%	9280
mgrid	15.5%	40	22.5%	14905
applu	15.1%	55	16.0%	27180
turb3d	27.6%	45	30.1%	5694
apsi	11.5%	65	13.0%	10813
wave5	23.4%	80	24.2%	7744
average (benchmarks)	22.7%	58	25.9%	11195

(a) Alpha platform

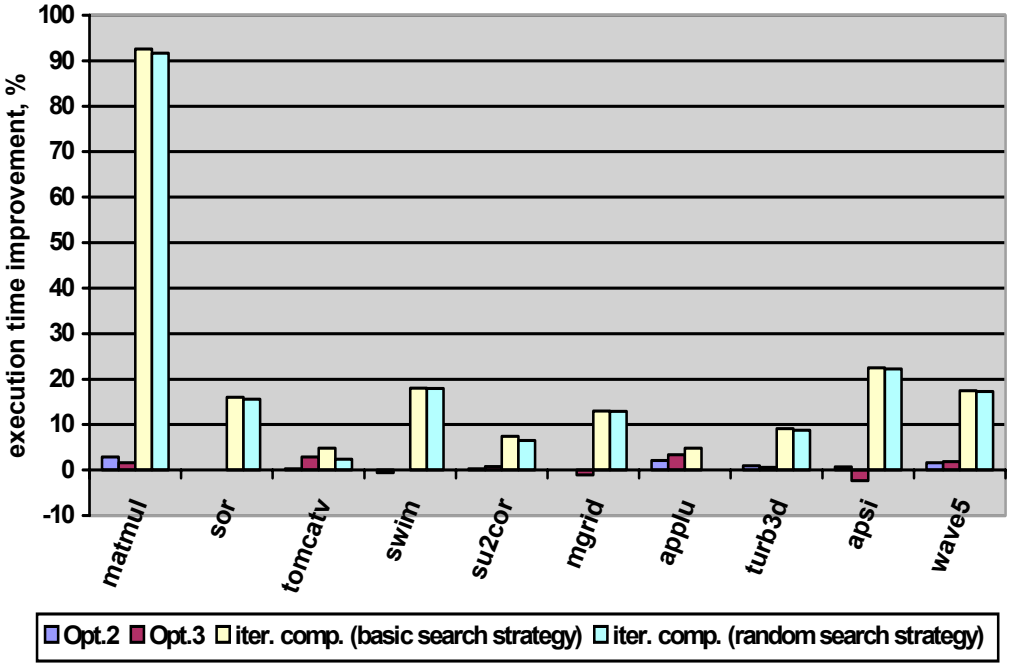
	Random search strategy		Basic search strategy	
	Performance improvement:	Number of iterations:	Performance improvement:	Number of iterations:
matmul	91.7%	20	92.6%	1599
sor	15.6%	20	16.0%	1599
average (kernels)	53.7%	20	54.3%	1599
tomcatv	2.4%	80	4.8%	7738
swim	17.9%	50	18.0%	6205
su2cor	6.5%	50	7.4%	9280
mgrid	12.9%	40	13.0%	14905
applu	0%	55	4.8%	27180
turb3d	8.8%	45	9.1%	5694
apsi	22.2%	65	22.5%	10813
wave5	17.3%	80	17.4%	7744
average (benchmarks)	11.0%	58	12.1%	11195

(b) Pentium platform

Table 6.5: Execution time improvements (%) and number of iterations needed after iterative compilation with the random and basic search strategies over Opt.1



(a) Alpha platform



(b) Pentium platform

Figure 6.2: Execution time improvements (%) after iterative compilation with the random search strategy and comparison with the basic search strategy and compiler optimisations

As in the case of iterative compilation with the basic search strategy, the random search strategy is also capable of achieving high performance improvements on kernels with a few loops and arrays. Moreover, the random search strategy needs only 20 iterations to obtain considerable performance improvements in contrast with 1599 iterations needed for the basic search strategy. There is less than 1% difference between performance improvements of matmul for both search strategies on two platforms. Figure 4.11 from chapter 4 explains this result. It shows that graphs with changes in execution time during each iterative step for matmul on both platforms are relatively smooth and therefore there is a high probability of choosing optimal transformation factors after only a few random tries. However, the performance improvement of sor drops considerably on the Alpha platform after the random search strategy by approximately 17% in comparison with the improvement after the basic search strategy. In contrast, the same benchmark has only a negligible reduction in the performance improvement on the Pentium platform. This can be explained by the fact, that the impact of transformations is greater and the performance improvements are higher on the Alpha platform than on the Pentium platform due to architectural features as demonstrated in section 4.5. Hence, the drop in the overall performance is also higher on the Alpha platform. On average there is around 9% reduction in the performance improvement on the Alpha platform and less than 1% reduction on the Pentium platform for kernels after the random search in comparison with the basic search. Nevertheless, it is a considerable performance improvement and is achieved only after 20 iterations in comparison with thousands of iterations of the basic search.

Performance improvements for SPEC benchmarks after iterative compilation with the random search strategy remain considerable and similar to ones after the basic search strategies on both platforms. For the Alpha platform, performance improvements vary between 11.5% and 43.3%. On the Pentium platform, applu is the only benchmark that does not have any performance improvement after the random search strategy that is explained by its complex structure and non-perfectly nested loops operating on five-dimensional matrices, which are hard to transform. Even using the basic search strategy, applu achieved only a small performance improvement of 4.8% after 27180 iterations. Therefore, using the random search

strategy with only 55 iterations is simply not enough to improve its performance. Nevertheless, all other benchmarks on the Pentium platform have significant performance improvements of up to 22.2%. Furthermore, the number of iterations needed to obtain such results is reduced considerably. It varies between 40 and 80 in contrast with the variation between 5694 and 27180 needed for the basic search strategy. On average, iterative compilation with the random search strategy performed well on both platforms with a small drop in performance improvement from 25.9% to 22.7% on the Alpha platform and from 12.1% to 11.0% on the Pentium platform.

When compared to Opt.2 and Opt.3 that are static and feedback-directed native compiler optimisations respectively, the random search strategy achieves also slightly less performance improvement than after the basic search strategy. However, it is still considerable for most of the programs on both platforms with the exception of tomcatv and applu on the Pentium platform. These benchmarks have already negligible performance improvements after iterative compilation with the basic search strategy due to various reasons described in section 4.5, and the native compiler manages to slightly outperform the random search strategy. Nevertheless, on average the performance improvements are considerable of 19.0% and 17.4% over Opt.2 and Opt.3 respectively on the Alpha platform and of 10.4% and 10.2% on the Pentium platform.

The experimental results obtained in this section show that using the performance prediction technique and the random search strategy dramatically reduces the number of iterations by about two orders of magnitude (370 times less in the case of mgrid) while still achieving considerable performance improvement comparable to the improvement obtained using the long basic search strategy. Besides, the random search strategy still outperforms the native compiler with either static or feedback-assisted optimisations for most of the programs on both platforms without any knowledge of the underlying hardware and with a minimal knowledge about the program structure. Furthermore, the small number of iterations is needed to achieve such performance improvements demonstrates the possibility of using iterative compilation not only in narrow areas such as optimising kernels but also for

```
algorithm FindB(N,C: integer) return integer;
    addr, di, dj, maxWidth: integer;

    maxWidth := min(N, C);
    addr := N/2;
    while true do
        addr: = addr + C;
        di := addr div N;
        dj := abs((addr mod N) – N/2);
        if di ≥ min(maxWidth, dj) then
            return min(maxWidth, di);
        maxWidth := min(maxWidth, dj);
    end while;
end algorithm;
```

Figure 6.3: Algorithm to compute the best tile size that removes self-interferences (Lam et al.)

optimising general-purpose, time-consuming applications on rapidly evolving hardware.

6.5 Comparison with existing techniques

Although iterative compilation outperforms existing commercial compilers, it should also be compared against published state-of-the-art static techniques. Here it is compared to two well-known static optimisation techniques proposed by Lam et al. in [LRW91] and by Coleman and McKinley in [CM95]. Both techniques are briefly reviewed in section 3.2.3. These methods attempt to reduce conflict and capacity misses by using loop tiling.

The algorithm for the first technique is presented in figure 6.3. It determines the largest square tile size that removes self-interference misses based on the periodicity in the addressing of a direct-mapped cache and the constant-stride accesses. This algorithm takes the matrix size N and the cache size C as the input and returns the largest tile size without conflict misses. The self-interference occurs between those array elements that are mapped into the same location in the cache and depends only

```
procedure TSS(CS, CLS, N, M)
        Input:    CS: cache size,  CLS: cache line size,
                  N: column length, M: row length
        Output:   tile size = bestCol * bestRow
bestCol = oldCol =  N
bestRow = rowSize = CS / N
colSize = CS - bestCol * bestRow
while (colSize > CLS & oldCol mod  colSize ≠ 0 & rowSize < M )
        rowSize = computeRows (colSize)
        tmp = colSize adjusted to a multiple of CLS
        if ( WSet (tmp, rowSize) > WSet (bestCol, bestRow)
                & WSet (tmp, rowSize) < CS
                & CIR (tmp, rowSize) < CIR (bestCol, bestRow)
                        bestCol = tmp
                        bestRow = rowSize
        endif
        tmp = colSize
        colSize = oldCol mod  colSize
        oldCol = tmp
endwhile
if necessary, adjust bestCol to meet the working set size constraint
end TSS
```

Figure 6.4: Algorithm to compute the best rectangular tile size (Coleman and McKinley)

on the difference of their addresses. Therefore, for a given array Y[i,j], the algorithm attempts to find array words Y[*di, N/2* ± *dj*] that are mapped to the same cache location. The returned largest best tile is the maximum of *di* and *dj*.

The second technique presented in figure 6.4 determines rectangular tiles to remove both capacity and conflict misses based on making the working set of the loop nest smaller than the cache size and on minimising cross interference misses for the tiled nests. This technique assumes that the cache is direct-mapped. As an input, the algorithm has the cache size (*CS*), the line size (*CLS*) and the array column

dimensions (*N* and *M*). To avoid self interference, the algorithm defines sets of consecutive columns of the array whose starting position differ by *N*. It further calculates the number of complete columns that fit into cache. The Euclidean algorithm is used to generate all potential column dimensions relatively fast. Initially the tile column size is set to *N* and than it is decreased until additional columns incur no interference. The column sizes are always selected as multiples of the cache line size to take advantage of spatial locality. To minimise cross interference, the new tile size is selected in such a way that the size of the working set is larger than for the previous tile size but it still fits in the cache while the cross interference rate is lower for the new tile size.

Both techniques can be applied to optimise blocked algorithms. Therefore, they are evaluated and compared with the developed iterative optimisation methods on the Alpha and Pentium platforms using two well-studied kernels with blocked algorithms: matmul and sor. The source codes of these kernels are presented in figure 4.2. Table 6.6 show the tile sizes selected for matmul and sor by the above techniques and by iterative compilation with the basic and random search strategies. Since both the above techniques apply only loop tiling, iterative compilation is restricted to loop tiling to have a fair comparison. The tile sizes selected by the algorithms presented in this section vary across two platforms. However, these sizes are the same for the same kernels as both kernels have the same array size. Furthermore, iterative compilation selected completely different tile sizes for both programs on both platforms.

Table 6.7 shows the performance improvements after applying the above optimisation algorithms to matmul and sor on the Alpha and Pentium platforms. These results show that Lam et al. and Coleman and McKinley algorithms perform reasonably well on matmul and considerably outperform the native compiler with both static and feedback-assisted optimisations on both platforms. This is explained by the fact that though matmul is a well-known kernel and is relatively easy to optimise as shown in detail in section 4.3, the native compilers appear to be concerned with avoiding degradation of performance after applying optimisations. Therefore, they either do not apply loop tiling or apply it with a small factor that generally does not degrade performance but improvements are also small. Hence, the

	matmul	sor
Lam et al. optimisation algorithm	16x16	16x16
Coleman and McKinley algorithm	512x16	512x16
Iterative compilation, basic search strategy (only tiling, 512 iterations)	8x8	4x4
Iterative compilation, random search strategy (only tiling, 5 iterations)	12x12	7x7

(a) Alpha platform

	matmul	sor
Lam et al. optimisation algorithm	8x8	8x8
Coleman and McKinley algorithm	512x8	512x8
Iterative compilation, basic search strategy (only tiling, 512 iterations)	67x67	5x5
Iterative compilation, random search strategy (only tiling, 5 iterations)	39x39	3x3

(a) Pentium platform

Table 6.6: Comparison of tile size selection by 4 algorithms: Lam et al., Coleman and McKinley, iterative compilation with the basic and random search strategies.

above static techniques have a greater potential to pick better tile size since the performance improvement graph as a function of tiling factors presented in section 4.3.2 for both platforms shows large flat minimum areas. Furthermore, even if the above static techniques fail to select the best blocking factor, they still gain considerable performance improvement. However, this is not the case for the sor, where both static optimisation algorithms and native compiler optimisations fail to achieve any performance improvement mainly due to assuming the use of the direct-mapped cache that is not the case and by using approximations to count interferences. Nevertheless, iterative compilation with the long basic strategy and with only loop tiling transformation enabled still outperforms both the above static optimisations and the native compiler, while having no knowledge of the targeted platform and having a minimum knowledge of the application. Furthermore, iterative compilation with the random search strategy outperforms the above techniques after only 5 iterations. Finally, iterative compilation with the random search strategy and with all transformations enabled considerably outperforms all the above methods and thus is a superior platform-independent optimisation method that can be applied to a wide range of programs.

	matmul	sor
Lam et al. optimisation algorithm	56.1%	0%
Coleman and McKinley algorithm	51.3%	0%
Native compiler static optimisations	31.2%	0%
Native compiler feedback-assisted optimisations	31.2%	0%
Iterative compilation, basic search strategy (only tiling, 512 iterations)	65.6%	25.4%
Iterative compilation, random search strategy (only tiling, 5 iterations)	63.8%	4.1%
Iterative compilation, basic search strategy (all transformations, 1599 iterations)	80.1%	28.6%
Iterative compilation, random search strategy (all transformations, 20 iterations)	79.2%	11.2%

(a) Alpha platform

	matmul	sor
Lam et al. optimisation algorithm	67.9%	0%
Coleman and McKinley algorithm	73.3%	0%
Native compiler static optimisations	2.9%	0%
Native compiler feedback-assisted optimisations	1.6%	0%
Iterative compilation, basic search strategy (only tiling, 512 iterations)	85.8%	5.8%
Iterative compilation, random search strategy (only tiling, 5 iterations)	85.4%	2.3%
Iterative compilation, basic search strategy (all transformations, 1599 iterations)	91.7%	16.0%
Iterative compilation, random search strategy (all transformations, 20 iterations)	91.2%	15.6%

(a) Pentium platform

Table 6.7: Execution time improvements (%) after static optimisation algorithms, after native compiler static and dynamic optimisations, after iterative compilation with loop tiling and after iterative compilation with all transformations enabled

6.6 Using smaller dataset

Using the performance prediction technique and the random search strategy as described above dramatically reduces the optimisation time by reducing the number of iterations. Potentially, there is another distinct method for reducing the overall optimisation time by using smaller datasets for a program during iterative compilation so that each program variant consumes less time. After iterative

Transformation: Dataset:	array padding	loop tiling	loop unrolling
256x256	1	not found	not found
512x512	1	16	not found
1024x1024	2	16	not found

Table 6.8: Best transformation factors that reduce execution time, found after iterative compilation with the basic search strategy for matmul with different datasets on the Alpha platform

optimisation is finished, the best transformation parameters found are used for the program with the original dataset. However, the main problem with this method is that different datasets can change the behaviour of the program dramatically and, therefore, the set of transformation factors best for the program with the smaller dataset is not necessarily the best for the same program with the original dataset. To demonstrate this issue, table 6.8 presents the best transformation factors found after applying iterative compilation with the basic search strategy for the matmul kernel on the Alpha platform using three distinct datasets with array sizes of 256x256, 512x512 and 1024x1024.

Consider three optimisation cases. The first one is when the dataset with the array size of 256x256 is used during iterative compilation with the basic search strategy to optimise the matmul kernel that further uses the dataset with the array size of 512x512. The second case is when the dataset with the array size of 512x512 is used to optimise the same kernel that further uses the dataset with the array size of 1024x1024. The last case is when the dataset with the array size of 256x256 is used to optimise matmul that further uses the dataset with the array size of 1024x1024. For the first case, the best array padding factor is the same, but the best loop tiling factor is different. For the second case, the best array padding is different, but the best loop tiling factor is the same. For the third case, both best array padding and loop tiling factors are different. In all cases, the best loop unrolling factor that could reduce execution time is not found. This results show that the sets of transformations are indeed different for different datasets.

Dataset:	Performance improvement (optimisation with the same dataset)	Performance improvement (optimisation with the smaller dataset)
512x512	80.1%	38.7% (256x256 dataset for optimisation)
1024x1024	86.4%	85.7% (512x512 dataset for optimisation)
1024x1024	86.4%	30.1% (256x256 dataset for optimisation)

Table 6.9: Comparison of performance improvements after iterative compilation with the basic search strategy for matmul when the original and smaller datasets are used during optimisation on the Alpha platform

Table 6.9 compares performance improvements after iterative compilation with the basic search strategy for the matmul kernel on the Alpha platform when both original and smaller dataset are used during optimisation. This table shows that the performance improvement dropped considerably in the first case from 80.7% to 38.7% and in the second case from 86.4% to 30.1%. However, the difference between performance improvements in the second case is negligible of 86.4% versus 85.7%.

Now, consider the iterative optimisation of the two SPEC benchmarks, swim and mgrid, with the basic search strategy using smaller training datasets to find good optimisation and then applying the resulting best optimisation to the actual reference data. Table 6.10 presents performance improvements for these benchmarks on both the Alpha and Pentium platforms. The results demonstrate that the outcome of the program optimisation with a smaller dataset also depends heavily on the platform used. For example, swim has a considerable performance improvement of 38.6% on the Alpha platform when using training dataset during optimisations while on the Pentium platform there was no any improvement. On the contrary, mgrid has a

	Alpha platform	Pentium platform
swim	38.6%	0%
mgrid	5.1%	9.6%

Table 6.10: Performance improvements after iterative compilation with the basic search strategy for SPEC benchmarks when the training dataset is used during optimisation and then the best optimisation is applied to the reference data

higher performance improvement on the Pentium platform than on the Alpha platform.

These results show that the smaller datasets can be potentially used for optimising programs using iterative compilation, however the drop in performance improvement can be significant in some cases. Therefore, more analysis is needed for the influence of different datasets on the program behaviour and optimisations such as in paper [EVD02] by Eeckhout et al, for example, where different datasets for a given program are analysed and various program-input pairs that are close to each other are selected to span the complete workload space. This is the topic of the future research.

6.7 Summary

This chapter presents methods to reduce the iterative search space dramatically, whilst still considerably outperforming current static optimisation methods and native compiler static and feedback-directed optimisations. First, performance prediction technique is used to remove those loops from the search space that have a negligible execution time or do not have a potential for further improvement. Second, a new search strategy is applied that tries only a small number of random factors for transformations instead of all possible ones. This reduces the number of iterations by two orders of magnitude without sacrificing performance much, thus making iterative compilation a realistic optimisation approach for a wide range of applications.

The results are compared with the performance improvements obtained using the native compilers and two well-known static optimisation techniques by Lam et al.

and Coleman and McKinley. Another method that reduces the iterative compilation time by using smaller datasets during program optimisation is also briefly examined. Experimental results show that this method can obtain considerable performance improvements on some datasets while gaining no speed-up on others. Therefore, it shows that there is a potential for reduction in compilation time but it requires further analysis of the influence of various datasets on program performance and is the topic of future research.

Chapter 7

Conclusions

This chapter briefly summarises the main results of this thesis, provides a critical review and proposes future work.

7.1 Summary

This thesis presented an automatic iterative compilation method for optimising numerical applications where memory latency is the dominant overhead. This platform-independent approach is based on feedback-directed program transformations. It is capable of outperforming considerably current well-known static and feedback-directed optimisations on large real applications. Moreover, iterative compilation never degrades program performance unlike other current methods that may degrade performance significantly. However, the major drawback of this method is the excessive compilation time where thousands of iterations are needed to achieve performance improvement. This thesis presented two techniques to reduce this time. First, a simple, fast and accurate performance prediction technique has been presented, that is capable of obtaining the lower bound on execution time if all cache misses were to be removed by transforming all program array references into scalars yet ensuring correct code execution. This technique can be used to considerably reduce the search space of iterative compilation by removing those loops from it that do not have any potential for improvement. It can also help programmers detect program sections that have a memory problem and therefore have to be optimised. Second, a random search strategy has been developed. This strategy tries only a small number of random factors for each transformation instead of all possible ones thus reducing the number of iterations by two orders of magnitude without significantly sacrificing performance.

A complete software toolset for automatic program analysis, transformations and optimisations has been created. It currently supports two distinct platforms: the Compaq Alpha with Digital Unix (RISC architecture) and the Intel Pentium with

Microsoft Windows (CISC architecture). The influence of array padding, loop tiling and loop unrolling on program performance has been analysed in detail on these two platforms. Furthermore, 2 kernels and 8 large SPEC benchmarks have been analysed and optimised using the developed iterative compilation strategy. Considerable performance improvements have been achieved in most of the cases in comparison with native state-of-the-art compilers and with well-known static optimisation techniques.

Therefore, the proposed iterative compilation approach with performance prediction and random search becomes a realistic platform-independent optimisation approach for a wide range of applications.

7.2 Critical review and future work

One of the drawbacks of the current implementation of iterative compilation presented in this thesis is that it is applied to programs with the same dataset size and with no conditional dependencies on the data values. To overcome this problem, program can be optimised several times for some typical datasets with the most time consuming branches taken. Further, conditional checks on the dataset can be embedded into the final program to choose different optimised versions. This will be the subject of future research.

The iterative optimisation method currently uses only three program transformations: array padding, loop tiling and loop unrolling. However, other transformations exist that can considerably improve performance: software pipelining, prefetching, loop distribution and fusion, for example. This will be implemented in the future. Since the above transformations can be used to optimise programs for ILP, they may be used for a wide range of programs, not only numerical applications with a memory problem.

The current implementation of iterative compilation uses source-to-source program transformations that can potentially interfere with the internal compiler optimisations and thus reduce the performance improvements. Therefore, the subject of future research is to analyse these interferences and to implement program transformations on the assembler level, preferably inside the compiler.

Though all the developed techniques are platform and language independent, the current software implementation is limited to two platforms and supports only Fortran transformations. In the future, other languages will be supported such as C, C++, Fortran 90 or even Java where iterative compilation engine could be embedded into just-in-time compiler to optimise programs at run time in the background. New platforms will be also supported in the future such as various DSPs or EPIC architectures.

One of the drawbacks of the performance prediction technique is that it is currently unable to fully handle programs with branches whose outcome depends on array values. This is the matter of the ongoing research and potentially can be handled by recording the frequency of the branch taken or by excluding these instructions from the prediction transformation.

The performance prediction technique provides a lower bound of the program execution time if all cache misses are removed. It will be combined with the memory and cache throughput so that it could not only predict the potential performance improvement but it could drive transformations to balance the calculations and memory access within a loop. It can work in a similar way to that described in [CK94] and [CG97] but is more precise and platform-independent as it does not require any approximations and simulations.

The performance prediction technique will be useful in analysing and optimising programs and can be implemented inside a production compiler as a profiling or feedback-directed optimisation option.

Finally, search strategies for iterative compilation will be improved to reduce compilation time even further by using current static and dynamic approaches to predict best transformation parameters and then to use them as a basis for a guided search strategy. The possibility to apply multiple transformations for various program sections in one step will be investigated as it can also reduce search time. In order to predict the overall iterative compilation time a set number of iterations will be used that can be spent on optimising the whole program. These iterations should be redistributed between sections of the program in such a way, that more iterations are used for the parts of the program where the potential for the improvement is higher.

Appendix A

Description of platforms

A.1 Alpha platform

Processor:	Digital Alpha 21264
Core frequency:	500 MHz
Bus frequency:	200 MHz
L1 data cache	
size:	64 KB
associativity:	2-way
line size:	64 bytes
L1 code cache	
size:	64 KB
associativity:	2-way
line size:	64 bytes
L2 cache	
size:	2048 KB
associativity:	direct-mapped
frequency:	200 MHz
line size:	64 bytes
bus width:	128 bits
Main memory:	512 MB
OS:	Digital Unix V4.0E (Rev. 1091)
Fortran:	Digital Fortran 77 Driver V5.2-10
	Digital Fortran 77 V5.2-171-428BH
C:	DEC C V5.8-009
Java:	Sun JDK 1.1.6-2

A.2 Pentium platform

Processor:	Intel Pentium III E
Core frequency:	650 MHz
Bus frequency:	100 MHz
L1 data cache	
size:	16 KB
associativity:	4-way
line size:	32 bytes
L1 code cache	
size:	16 KB
associativity:	4-way
line size:	32 bytes
L2 cache	
size:	256 KB
associativity:	8-way
frequency:	650 MHz
line size:	32 bytes
bus width:	256 bits
Main memory:	192 MB
OS:	Windows 2000 Professional (SP3)
Fortran:	Intel Fortran 6.0 Build 020321Z
C:	Intel C 6.0 Build 020321Z
Java:	Sun JDK 1.1.8

Bibliography

[ABD+97] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S-T.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Transactions on Computer Systems,* 15(4), pages 357-390, November 1997.

[AI91] C. Ancourt and F. Irigoin. Scanning Polyhedra with DO Loops. *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39-50, April 1991.

[AJL+95] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3), pages 367-432, September 1995.

[ALE02] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modelling. *IEEE Computer*, 35(2), pages 59-67, February 2002.

[AMP00] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. *Proceedings of the IEEE/ACM Conference on Supercomputing* (SC'2000), November 2000.

[AP93] A. Agarwal and S.D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. *Proceedings of the 20th annual international symposium on Computer architecture (ISCA)*, pages 179-190, May 1993.

[ASU86] A. Aho, R. Sethi, and J.D. Ullman. *Compilers – principles, techniques and tools*. Addison-Wesley, 1986.

[ATA+00] J. Abella, S.A.A. Touati, A. Anderson, C. Ciuraneta, J.M. Codina, M. Dai, C. Eisenbeis, G. Fursin, A. Gonzalez, J. Llosa, M. O'Boyle, A. Randrianatoavina, J. Sanchez, O. Temam, X. Vera, and G. Watts. The MHAOTEU toolset for memory hierarchy management. *IMACS'2000, 16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, August 2000.

[BAB96] D. Burger, T.M. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. University of Wisconsin Computer Sciences Technical Report CS-TR-1996-1308. July 1996.

[Ban88] U.K. Banerjee. Dependence analysis for supercomputing. Kluwer Academic Publishers, Norwell, MA, 1988.

[BDG02] M. Butts, A. DeHon, S.C. Goldstein. Molecular electronics: devices, systems and tools for gigagate, gigabit chips. *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design (ICCAD)*, pages 433-440, November 2002.

[BG97] D. Burger and J.R. Goodman. Billion-transistor architectures. *IEEE Computer*, 30(9), pages 46-48, September 1997.

[BGN63] A.W. Burks, H.H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. *John von Neumann Collected Works,* Vol. 5, A. H. Taub Editor, The Macmillian Co., New York, pages 34-79, 1963.

[BGS94] David F. Bacon, Susan L. Graham, and Olivier J. Sharp. Compiler transformations for high-performance computing. ACM Computing Surveys, 26(4), pages 345-420, 1994.

[BH00] B.R. Buck and J.K. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. *Proceedings of the IEEE/ACM Conference on Supercomputing (SC'2000),* November 2000.

[BJW+92] F. Bodin, W. Jalby, D. Windheiser, and C. Eisenbeis. A quantitative algorithm for data locality optimization. *Code Generation: Concepts, Tools, Techniques*, Springer-Verlag, pages 119-145, 1992.

[Blo59] E. Bloch. The engineering design of the Stretch computer. *Proceedings of the Eastern Joint Computer Conference*, pages 48-59, 1959.

[BS95] F. Bodin and A. Seznec. Skewed associativity enhances performance predictability. *Proceedings of the 22th Annual International Symposium on Computer Architecture (ISCA)*, pages 265-274, June 1995.

[CG97] S. Carr and Y. Guan. Unroll-and-jam using uniformly generated sets. *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 349-357, December 1997.

[CJD+01] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. High-performance DRAMs in workstation environments. *IEEE Transactions on Computers*, 50(11), pages 1133-1153, November 2001.

[CK94] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6), pages 1768-1810, November 1994.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 40-52, April 1991.

[CL99] R. Cohn and P.G. Lowney. Feedback directed optimization in Compaq's compilation tools for Alpha. *Proceedings of the 2nd ACM Workshop on Feedback-Directed Optimization*, pages 3-12, November 1999.

[CM95] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI),* pages 279-290, June 1995.

[CMH91] P.P. Chang, S.A. Mahlke, and W.W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12), pages 1301-1321, 1991.

[DH79] J.J. Dongarra and A.R. Hinds. Unrolling loops in Fortran. *Software Practice and Experience*, 9(3), pages 219-226, 1979.

[DHP+77] J. Davidson, W. Hathaway, J. Postel, N. Mimno, R. Thomas, and D. Walden. The arpanet telnet protocol: Its purpose, principles, implementation, and impact on host operating system design. *ACM Proceedings of the 5th Symposium on Data Communications*, pages 4.10-4.18, September 1977.

[DHW+97] J. Dean, J.E. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. *Proceedings of the 30st Annual IEEE International Symposium on Microarchitecture,* pages 292-302, December 1997.

[Dun90] R. Duncan. A survey of parallel computer architectures. *IEEE Computer*, 23(2), pages 5-16, February 1990.

[EVD02] L. Eeckhout, Hans Vandierendonck, and De Bosschere. Workload design: selecting representative program-input pairs. IEEE Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'02), pages 83-94, 2002.

[Fea92] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part {II}. Multidimensional time. *International Journal of Parallel Programming*, 21(6), pages 389-420, 1992.

[FHM+96] F. Faggin, M. Hoff Jr., S. Mazor, and M. Shima. The history of the 4004. *IEEE Micro*, 16(6), pages 10-20, December 1996.

[Fis83] Joseph A. Fisher. Very Long Instruction Word architectures and the ELI-512. *Proceedings of the 10th annual international symposium on Computer architecture (ISCA)*, pages 140-150, June 1983.

[Fly72] M.J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, C-21(9), pages 948-960, September 1972

[Fra02] M.P. Frank. The physical limits of computing. *IEEE Computing in Science & Engineering*, 4(3), pages 16-26, May/June 2002.

[GH88] J.R. Goodman and W.C. Hsu. Code scheduling and register allocation in large basic blocks. *Proceedings of the 2nd International Conference on Supercomputing (ICS)*, pages 442-452, 1988.

[GKM82] S.L. Graham, P.B. Kessler, and M.K. McKusick. Gprof: A call graph execution profiler. *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120-126, June 1982.

[GMM98] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 228-239, October 1998.

[HP96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HMR+00] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, R. Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5), pages 12-23, September 2000.

[Int03a] Intel® Itanium® 2 processor datasheet. *Document number 250945-002*, June 2003, http://developer.intel.com/design/itanium2

[Int03b] Intel Corporation. VTune performance analyzer. Website: http://developer.intel.com/software/products/vtune/index.htm, 2003.

[Joh91] M. Johnson. *Superscalar microprocessor design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[KCR+98] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 285-297, 1998.

[KE62] T. Kilburn, D.B.C. Edwards, M.I. Lanigan, and F.H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2), pages 223-235, April 1962.

[KF03] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4), pages 500-548, July 2003.

[KKO+00] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and H.A.G. Wijshoff. Iterative compilation in program optimization. *Proceedings of the 8th International Workshop on Compilers for Parallel Computers (CPC'2000)*, pages 35-44, January 2000.

[Lam88] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI)*, pages 318-328, June 1988.

[LL97] A.W. Lim and M.S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 201-214, January 1997.

[LLL01] A.W. Lim, S-W. Liao, and M.S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 103-112, 2001.

[Llo00] S. Lloyd. Ultimate physical limits to computation. *Nature 406*, pages 1047-1054, August 2000.

[LP92] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science 757,* Springer-Verlag, pages 391-405, 1992.

[LRW91] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63-74, April 1991.

[Lun02] M. Lundstrom. Is nanoelectronics the future of microelectronics? *Proceedings of the 2002 International Symposium on Low Power Electronics and Design (ISLPED)*, pages 172-177, August 2002.

[LW94] A.R. Lebeck and D.A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10), pages 15-26, October 1994.

[MB76] A. Madison and A. Batson. Characteristics of program localities. *Communications of the ACM*, 19(5), pages 285-294, May 1976.

[MRB+99] P. van der Mark, E. Rohou, F. Bodin, Z. Chamski, and C. Eisenbeis. Using iterative compilation for managing software pipeline-unrolling trade-offs. *Proceedings of SCOPES'99*, 1999

[MCT96] K.S. McKinley, S. Carr, and CW. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4), pages 424-453, July 1996.

[MLG92] T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pages 62-73, October 1992.

[MN03] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2), pages 44-55, March-April 2003.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), pages 114-117, April 1965.

[MT96] K.S. McKinley and O. Temam. A quantitative analysis of loop nest locality. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 94-104, October 1996.

[MT99] K.S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4), pages 288-336, November 1999.

[Nis98] A. Nisbet. GAPS: Genetic algorithm optimised parallelization. *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, 1998.

[NS97] D.B. Noonburg and J.P. Shen. A framework for statistical modeling of superscalar processor performance. *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture (HPCA)*, pages 298-309, February 1997.

[OK99] M.F.P. O'Boyle and P.M.W. Knijnenburg. Non-singular data transformations: definition, validity, applications. *International Journal on Parallel Programming*, 17(3), pages 131-159, 1999.

[PD80] D.A. Patterson and D.R. Ditzel. The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, 8(6), pages 25-33, October 1980.

[PH90] K. Pettis and R.C. Hensen. Profile guided code partitioning. *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*, pages 16-27, June 1990.

[PTV+92] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in Fortran: The Art of Scientific Computing.* Cambridge University Press, second edition, 1992.

[Pug91] W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 4-13, November 1991.

[RL77] C.V. Ramamoorthy and H.F. Li. Pipeline architecture. *ACM Computing Surveys*, 9(1), pages 61-102, March 1977.

[RP96] J.M. Rabaey and M. Pedram (Eds). *Low power design methodologies*. Kluwer Academic Publishers, 1996.

[RT98] G. Rivera and CW. Tseng. Data transformations for eliminating conflict misses. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 38-49, June 1998.

[RT99] G. Rivera and CW. Tseng. A comparison of compiler tiling algorithms. *Proceedings of the 8th International Conference on Compiler Construction*, pages 168-182, March 1999.

[Sar00] V. Sarkar. Optimized unrolling of nested loops. *Proceedings of the 14th International Conference on Supercomputing*, pages 153-166, May 2000.

[SCD+97] M. Schlansker, T.M. Conte, J. Dehnert, K. Ebcioglu, J.Z. Fang, and C.L. Thompson. Compilers for instruction-level parallelism. *IEEE Computer*, 30(12), pages 63-69, December 1997.

[Sch02] J.P. Scheible. A survey of storage options. *IEEE Computer*, 35(12), pages 42-46, December 2002.

[SE94] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI),* pages 196-205, June 1994.

[SG00] J. Sánchez and A. González. Analyzing data locality in numeric applications. *IEEE Micro*, 20(4), pages 58-66, July/August 2000.

[Sin92] A. Sinha. Client-server computing. *Communications of the ACM*, 35(7), pages 77-98, July 1992.

[Smi82] A.J. Smith. Cache memories. *ACM Computing Surveys*, 14(3), pages 473-530, September 1982.

[Smi91] M.D. Smith. Tracing with pixie. Technical report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, November 1991.

[Smi00] M.D. Smith. Overcoming the challenges to feedback-directed optimization. *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1-11, 2000.

[SPE03] The Standard Performance Evaluation Corporation. Website: http://www.specbench.org, 2003.

[SR00] M.S. Schlansker and B.R. Rau. EPIC: Explicitly parallel instruction computing. *IEEE Computer*, 33(2), pages 37-45, February 2000.

[TFJ94] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 261-271, May 1994.

[TG99] N. Topham and A. González. Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computers*, 48(2), pages 185-192, February 1999.

[TGJ93] O. Temam, E.D. Granston, and W. Jalby. To Copy or Not to Copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. *Proceedings of the 1993 ACE/IEEE Conference on Supercomputing*, pages 410-419, 1993.

[Tou02] Sid-Ahmed-Ali Touati. *Register pressure in instruction level parallelism*. Ph.D. thesis, Université de Versailles Saint-Quentin, June 2002.

[UM97] R.A. Uhlig and T.N. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 29(2), pages 128-170, June 1997.

[VE00] M.J. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. *Proceedings of the 2000 International Conference on Parallel Processing*, pages 163-172, August 2000.

[VKT+97] E. van der Deijl, G. Kanbier, O. Temam, and E.D. Granston. A cache visualization tool. *IEEE Computer*, 30(7), pages 71-78, July 1997.

[VL00] S.P. VanderWiel and D.J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2), pages 174-199, June 2000.

[VX02] X. Vera and J. Xue. Let's study whole-program cache behaviour analytically. *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 176-185, 2002.

[WD98] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. *Proceedings of the IEEE/ACM Conference on Supercomputing (SC'98)*, November 1998.

[WL91a] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI)*, pages 30-44, June 1991.

[WL91b] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), pages 452-471, October 1991.

[Wol89] M.J. Wolfe. More iteration space tiling. *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 655-664, November 1989.

[Xue97a] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12), pages 1621-1645, 1997.

[Xue97b] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4), pages 409-424, 1997.

[YBH01] Y. Yu, K. Beyls, E.H. Hollander. Visualizing the impact of the cache on program execution. *Proceedings of the 5th International Conference on Information Visualisation (IV'01)*, pages 336-341, July 2001.

[ZWG+97] X. Zhang, Z. Wang, N. Gloy, J.B. Chen, and M.D. Smith. System support for automatic profiling and optimization. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 15-26, October 1997.