SMART MEMORY MANAGEMENT THROUGH LOCALITY ANALYSIS

Jesús Sánchez

DEPT. OF COMPUTER ARCHITECTURE
UNIVERSITAT POLITÈCNICA DE CATALUNYA
Barcelona (SPAIN)

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor en Informática

ABSTRACT

Cache memories were incorporated in microprocessors in the early times and represent the most common solution to deal with the gap between processor and memory speeds. However, many studies point out that the cache storage capacity is wasted many times, which means a direct impact in processor performance. Although a cache is designed to exploit different types of locality, all memory references are handled in the same way, ignoring particular locality behaviors. The restricted use of the locality information for each memory access can limit the effectivity of the cache. In this thesis we show how a data locality analysis can help the researcher to understand where and why cache misses occur, and then to propose different techniques that make use of this information in order to improve the performance of cache memory. We propose techniques in which locality information obtained by the locality analyzer is passed from the compiler to the hardware through the ISA to guide the management of memory accesses.

We have developed a static data locality analysis. This analysis is based on reuse vectors and performs the three typical steps: reuse, volume and interfere analysis. Compared with previous works, both volume and interference analysis have been improved by using profile information as well as a more precise interference analysis. The proposed data locality analyzer has been inserted as another pass in a research compiler. Results show that for numerical applications the analysis is very accurate and the computing overhead is low. This analysis is the base for all other parts of the thesis. In addition, for some proposals in the last part of the thesis we have used a data locality analysis based on cache miss equations. This analysis, although more time consuming, is more accurate and more appropriate for set-associative caches. The usage of two different locality analyzers also shows that the architectural proposals of this thesis are independent from the particular locality analysis.

After showing the accuracy of the analysis, we have used it to study the locality behavior exhibited by the SPECfp95 programs. This kind of analysis is necessary before proposing any new technique since can

help the researcher to understand why cache misses occur. We show that with the proposed analysis we can study very accurately the locality of a program and detect where the hot spots are as well as the reason for these misses. This study of the locality behavior of different programs is the base and motivation for the different techniques proposed in this thesis to improve the memory performance.

Thus, using the data locality analysis and based on the results obtained after analyzing the locality behavior of a set of programs, we propose to use this analysis in order to guide three different techniques: (i) management of multi-module caches, (ii) software prefetching for modulo scheduled loops, and (iii) instruction scheduling for clustered VLIW architectures.

The first use of the proposed data locality analysis is to manage a novel cache organization. This cache supports bypassing and/or is composed of different modules, each one oriented to exploit a particular type of locality. The main difference of this cache with respect to previous proposals is that the decision of caching or not, or in which module a new fetched block is allocated is managed by some bits in memory instructions (locality hints). These hints are set at compile time using the proposed locality analysis. Thus, the management complexity of this cache is kept low since no additional hardware is required. Results show that smaller caches with a smart management can perform as well as (or better than) bigger conventional caches.

We have also used the locality analysis to study the interaction between software pipelining and software prefetching. Software pipelining has been shown to be a very effective scheduling technique for loops (mainly in numerical applications for VLIW processors). The most popular scheme for software pipelining is called modulo scheduling. Many works on modulo scheduling can be found in the literature, but almost all of them make a critical assumption: they consider an optimistic behavior of the cache (in other words, they use the hit latency when a memory instruction is scheduled). Thus, the results they present ignore the effect of stalls due to dependences with memory instructions. In this part of the thesis we show that this assumption can lead to schedules whose performance is rather low when a real memory is considered. Thus, we propose an algorithm to schedule memory instructions in modulo scheduled loops. We have studied different software prefetching strategies and finally proposed an algorithm that performs prefetching based on the locality analysis and the shape of the loop dependence graph. Results obtained shows that the proposed scheme outperforms other heuristic approaches since it achieves a better trade-off between compute and stall time than the others

Finally, the last use of the locality analysis studied in this thesis is to guide an instruction scheduler for a clustered VLIW architecture. Clustered architectures are becoming a common trend in the design of embedded/DSP processors. Typically, the core of these processors is based on a VLIW design which partitionates both register file and functional units. In this work we go a step beyond and also make a partition of the cache memory. Then, both inter-register and inter-memory communications have to be taken into account. We propose an algorithm that performs both graph partition and instruction scheduling in a single step instead of doing it sequentially, which is shown to be more effective. This algorithm is improved by adding an analysis based on the cache miss equations in order to guide the scheduling of memory instructions in clusters with the aim of reducing not only inter-register communications, but also cache misses.

AGRADECIMIENTOS

Me gustaría agradecer la confianza depositada en mí por Antonio González, mi director de tesis, el cual me ha guiado durante todo este camino. Gracias por su paciencia y por todas las cosas que me ha enseñado.

Gracias a toda mi familia: a mis padres, Paco y Juani, a mi abuela, Carmen, y a mi hermano, Carlos, y a todo el resto de familia que han creido en mi y me han apoyado durante todo este tiempo, y muy especialmente a Lorena, que ha aparecido en mi vida en el último año de tesis y que me ha iluminado desde entonces con su sonrisa. Te quiero, osita! ;-)

Agradecer el apoyo y ayuda de toda la gente del departamento que en algún u otro momento me han echado una mano. Prefiero no poner una lista de nombres porque estaríais todos, pero muy especialmente a mis amigos Profesor Titular Pepe González, con quien comencé toda esta aventura, y Manolo Fernández, por todos esos buenos ratos dentro y fuera de la Universidad.

Por último, no quisiera olvidarme de todos mis amigos fuera de la Universidad con los que he pasado tan buenos momentos y con los que siempre he conseguido olvidarme un poco de procesadores, compiladores y rollos de esos.

CONTENTS

LIST OF FIGURES

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

CHAPTER 6

CHAPTER 7

LIST OF TABLES

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

CHAPTER 6

CHAPTER 7

A mis padres,
Paco y Juani,
por todo su apoyo.

1

INTRODUCTION

Cache memories are fast and small memories that current microprocessors include in order to mitigate the gap between processor and memory speeds. The basic idea is to design a hierarchy of memories (one or several levels) between the processor and the main memory such that a memory access may be solved by one of these levels much faster than by the main memory. The effectiveness of this solution strongly depends on the proper usage of the cache hierarchy. The ability of a cache to store the most useful data is based on exploiting the locality exhibited by memory references. In this chapter we review different techniques, both hardware and software, to improve the behavior of cache memories, especially techniques focused on the first level cache, and then we introduce the work developed in this thesis.

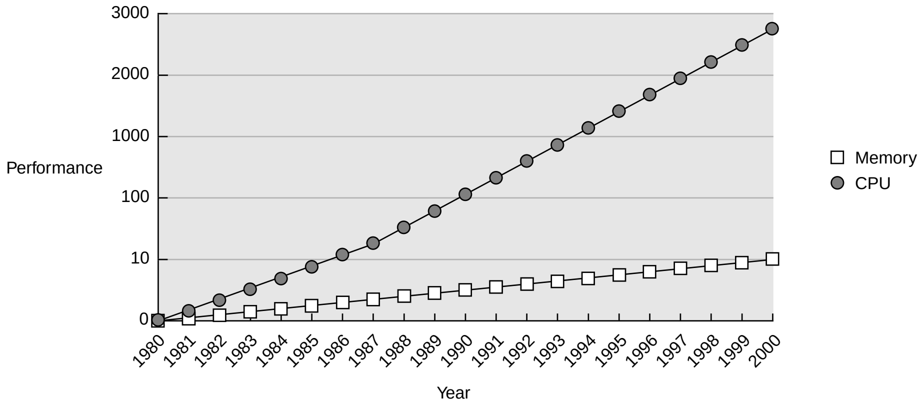


Figure 1.1. Evolution in the performance of memory and CPU speeds during last 20 years

1.1. MOTIVATION

One of the main problems that computer architects have to face is the gap between processor and memory speeds. Advances in the design of microprocessors have experienced notable improvements over the last 15 years and this tendency is likely to hold for the next years. The most important improvements have been done in both clock rates and execution parallelism. By reducing the minimum feature size, new technologies pack more logic in a single chip, and allow transistor to switch faster. Furthermore, *instruction-level parallelism* (ILP), exploited by both hardware and compiler technology, has also been key for this performance growth. However this trend has not been experienced by memory technology. Whereas microprocessor performance has improved 55% per year since 1987, and 35% per year until 1986, the improvement in memory latency is only a 7% per year. This discrepancy in the evolution of processor and memory performances can be seen in Figure 1.1., extracted from [43].

This discrepancy in speed has fostered many studies to mitigate this gap. The most extended solution is the inclusion of a memory hierarchy, and then the concept of *cache memory* is introduced [109]. Cache memories are small and fast buffers that are used to store recently-used information. The basic idea behind a memory hierarchy is to put a cache memory between the processor and the main memory. Thus, a memory access from the processor first tries in the cache, and if the information is found, the access is quickly solved. This allows the processor not to wait for the information to be brought from main memory. Otherwise, if the information is not found in cache, it is fetched from memory and temporary stored there. So that a possible future access to this information can be solved with minimum latency. Cache memories are usually provided for data and instructions separately in the first level. Memory hierarchies with two or three levels of cache between the processor and the main memory are common in existent computers. Current designs have the first level cache (and even the second) inside the same chip as the processor in order to achieve a low access latency.

The performance of this scheme highly depends on the ability of the cache to keep the information that will be used in a near future. Guesses about future references rely on the concept of *locality* [43]. The ability of the processor to exploit the locality depends on both the cache architecture and the reference patterns generated by the program.

As we will see in this chapter and during the rest of the thesis, many heuristics have been proposed in order to improve the performance of the basic cache architecture. Some of these techniques consist of code transformations or data reorganizations that are done by the compiler, whereas some other techniques rely on some additional hardware. We can also find in the literature some hybrid techniques that combine software and hardware mechanisms. All heuristics proposed are based on the exploitation of some particular locality features that have been observed in programs. However not all of them make use of a locality analysis that indicates when this heuristic has to be applied. It is shown in some works and in some parts of this thesis that a “blind” use of many of these heuristics can lead to an underutilization of the cache and can even degrade its performance. For instance, it is shown that an unconditional use of prefetching contributes to the pollution of the cache with useless blocks that can replace useful blocks. A better knowledge of the locality of programs, defining more precisely where the problems are and what are their causes is a key factor for an effective usage of many techniques.

In this thesis we show that data locality of programs can be accurately analyzed. For numeric applications, this analysis can be performed almost statically by the compiler, obtaining then a very fast and very accurate analysis. Moreover, the information that a static analysis can offer is very wide, identifying clearly the hot points in the program. This analysis is used to propose different alternatives in the management of the cache in order to efficiently use the storage space. We explore different techniques that “communicate” the locality information to some particular cache organizations, obtaining then techniques that, based on explicit locality information, make a smarter use of the bare cache scheme.

The organization of this chapter is the following one. First we will review some basic cache concepts as well as different classical techniques (both hardware and software) proposed to improve the performance of the basic cache scheme. Then, we will highlight the contributions of this thesis and finally the organization of the remaining chapters.

1.2. BACKGROUND

Typical caches are characterized by three different parameters: (i) capacity, (ii) block size, and (iii) associativity. The capacity of a cache corresponds to the amount of information (measured in bytes) that is able to store. This capacity is commonly divided into blocks (also referred as lines). A cache block is the amount of contiguous information that is brought, when necessary, from the next level of the hierarchy. Then, the block size is the size of one of these blocks. Finally, the associativity of a cache represents the number of different locations (in this case, blocks) in which a new block brought from the next hierarchy level can be stored. This parameter, thus, also indicates the number of positions in which we have to search for each access. If the associativity is one, the cache is called *direct-mapped*. On the other extreme, if a new block can be stored in any block of the cache, it is called *fully-associative*. Finally, intermediate configurations are called *n-way set-associative* caches, where *n* represents the associativity.

All current general-purpose processors are implemented with at least one cache level. Table 1.1 shows different examples with their features. Even in the scenario of embedded/DSP processors, that have typically been designed without a memory hierarchy, we can nowadays find commercial microprocessors with the inclusion of one or more levels of cache [102].

Family	Model	L1				L2 (inside the chip)	
		DATA		INSTRUCTIONS			
		Size (KB)	Assoc.	Size (KB)	Assoc.	Size (KB)	Assoc.
Intel	**Pentim II**	16	4	16	4	-	-
	Pentium III	16	?	16	?	256	8
AMD	**K7**	64	2	64	2	-	-
PowerPC	**620**	32	8	32	8	-	-
Sparc	**UltraSparc II**	16	2	16	1	-	-
Alpha	**21164**	8	1	8	1	96	?
	21264	64	2	64	2	-	-
	21364	64	2	64	2	1.5MB	6
MIPS	**R10000**	32	2	32	2	-	-
HP	**PA-7200**	-	-	64	2	-	-
	PA-7300	64	2	64	2	-	-
	PA-8500	1MB	4	0.5MB	4	-	-

Table 1.1. Cache hierarchy configuration in current microprocessors

Many techniques and proposals can be found in the literature to improve the behavior of a cache. These techniques could be grouped into three different kinds, depending on their aim:

- Reduce the penalty on a cache miss
- Reduce the number of misses
- Reduce the number of memory accesses

Among all these different techniques, some of them are implemented just by hardware so that the original code is not affected, whereas some other proposals are based on code transformations, with some possible support of the hardware. In the following subsections we review some of the most classical techniques. Due to the huge number of proposed techniques, this review will probably miss some techniques that some readers may consider very important. Other techniques more related to particular proposals of this thesis are reviewed in the following chapters where the proposals are presented.

1.2.1. Hardware-Based Techniques

Hardware approaches are implemented by modifying the organization of the cache core itself or by adding new submodules with a special goal. The main advantage of these schemes is that no requirement from the instruction set nor from the user/compiler is required.

Higher associativity

As many studies point out, one of the reasons why memory references miss in cache is due to the mapping function. In a *direct-mapped* cache, each block that is brought to the cache has just one possible location. This means that if several blocks that are used in the same interval of time are mapped onto the same location, each one will alternatively replace the other, provoking then cache misses. The basic solution to this problem is to increase the associativity of the cache. Some different algorithms can be found in order to select which one of the multiple locations is chosen if any block has to be replaced (the most common algorithm is called LRU - Least Recently Used).

However, the main drawback of increasing the associativity of a basic cache is that the complexity of the cache is increased as well. This fact produces an increment in both the access time and the area of the cache. Then, one of the roles of the designer is to choose the best trade-off among these terms. Typical organizations in modern high-performance processors are 2-way or 4-way set associative caches.

Larger block size

Another way to reduce the miss rate is to increase the block size of the basic cache. Larger block sizes take advantage of spatial locality and at the same time reduce compulsory misses. However, an important remark is the fact that a cache is designed to exploit an average of the temporal and spatial reuse. Note that temporal reuse does not benefit from larger block sizes but from having more cache blocks. With the same cache capacity, a larger block means less number of blocks, and then, we have here another trade-off to deal with. Typical cache blocks in modern microprocessors are 32 or 64 bytes.

Second-level caches

The most common way to reduce the penalty of a cache miss is to introduce one or two levels of caches between the first level cache and the main memory. These caches could be placed inside the processor chip or outside. An example of microprocessors with several levels of cache is the Alpha 21364 [40]. This processor has a 64KB L1 data cache, a 64KB L1 instruction cache and a 1.5MB L2 unified second cache on-chip, and a can have an L3 cache off-chip.

Write buffers

In a simple cache organization, the CPU must wait for stores if the block to be updated is not in cache. The simple policy is waiting until the block is fetched from the next memory level and then updated. A common optimization to reduce stalls due to store operation is the addition of a write buffer. A write buffer consists on a small (typically fully-associative) buffer in which write requests are temporary stored until they are written to cache or memory. It allows the CPU to wait just until the request has been written onto the write buffer, reducing then the store latency. Once the request is in the write buffer, the cache takes profit of inactive cycles to deal with entries in the buffer. On later cache accesses, both cache and write buffers are tried in parallel. If a block is in both the cache and the write buffer, the request is resolved by the latter, since it has the most recently updated block.

Victim caches

The *victim cache*'s [52] primary goal is to remove conflict misses. The basic idea is to have a small fully-associative module where blocks discarded from the main cache are placed. If a hit occurs in the victim cache, a swapping of blocks between the victim and the main cache is performed. A similar cache architecture is the PA-7200 *assist cache* [15]. The management of the two modules is somewhat different, the software-controlled selective swapping being its most important difference. Memory instructions in the PA-7200 have a flag that is set by the compiler for those instructions that are expected to exhibit only spatial reuse. The data accessed by these instructions are brought into the assist cache but are not later moved to the main cache.

Hardware prefetching

Data prefetching techniques basically consist of bringing memory blocks to cache before they are referenced. Basic hardware prefetching techniques appeared practically in conjunction with the earlier proposals of caches memories. In fact, a conventional cache implements implicitly a mechanism of hardware prefetching since on a miss, a memory block (instead of just a single data) is fetched, taking then profit of the spatial locality.

The first interesting hardware prefetching schemes were proposed by A. J. Smith [94]. He proposed a techniques called *one block lookahead* (*OBL*), that is, on an access to block *i*, the block *i+1* is prefetched, taking then profit from spatial locality. He also proposed alternative approaches to performing the prefetching: (1) always on a cache access, (2) on a cache miss, and (3) tagged prefetch. This last option works as follows: on a miss, the current and next blocks are fetched, and this last one is tagged with zero. When an access to a block with tag zero is performed, the tag is set to one, and the prefetch to the next line is performed (in this case, if the new line is not in cache, it is tagged with zero).

With the years, new hardware prefetching techniques have been proposed. *Lookahead schemes* try to solve the problem when the stride of the access is large. However, maybe the most well-known scheme are the *stream buffers*. S*tream buffers* were proposed by N. Jouppi [52]. They consist of FIFO queues added between the L1 and L2 caches where some consecutive memory blocks are stored. On an access to memory, both the L1 cache and the tops of each *stream buffers* are tried in parallel. If the access hits on the cache, the *stream buffers* remain untouched. However, on a miss in cache, if the access hits in a *stream buffer*, the corresponding block is moved to the cache. Then, the *stream buffer* prefetches the next block in the next level of the memory hierarchy. When there is miss in both the L1 cache and the *stream buffers*, a new *stream buffer* (if any) is reserved and the next blocks are prefetched. As the prefetched data are stored in buffers apart from the L1 cache, the scheme avoids the possible pollution contributed by the prefetched data. Nevertheless, *stream buffers* behave well for small strides and when programs do not deal at the same time with more structures (e.g., arrays or matrices) than available *stream buffers*.

Non-blocking caches

On a typical memory access, a miss on the first level cache provokes all the CPU to stall until the data is returned to the target register, even if this data is not used immediately.

Non-blocking caches where originally proposed by Kroft [60] and since then, some studies about the possible implementation [26] and its impact in the performance [16] have been proposed. The main idea behind a non-blocking cache (also known as lockup-free cache) is that instructions that miss do not stall the system, that is, the memory access will be served in parallel with the execution of subsequent instructions, including other memory accesses, that do not need the data. Moreover, a common feature in non-blocking caches is the possibility of dealing with multiple outstanding misses concurrently.

Non-blocking caches are very common in current processors. Examples of processors that use them are the MIPS R1000, the PowerPC 620, the HP-PA8000 or the Alpha 21164.

1.2.2. Software-Based Techniques

Another family of techniques are those performed at compile time. In this case, the compiler is the responsible for transforming the code with the goal of taking the maximum profit to the memory hierarchy.

High-level code transformations

High-level code transformations aim to restructure some parts of the code in order to increase the memory performance. These transformations can be performed in the high-level representation of the code. Each technique is typically oriented to exploit a particular feature of the memory access patterns, and then, a typical optimization pass is composed of several of these transformations. Some of the more common are:

- *Loop interchange*: it consists of exchanging the position of two loops in a loop nest by moving one of the outer loop to inner positions. This transformation can improve the performance in many different ways. Regarding the improvement of the data locality, it can help to reduce the stride of the access (ideally to stride one), and then exploiting spatial locality in a cache block.
- *Loop blocking (or tiling)*: this transformation helps to improve the locality of accesses when it is limited by the cache capacity. Blocking is accomplished by modifying the order in which the iteration space is traversed so that reuses of data occur at a shorter distance in time. Thus, the storage requirements to exploit the locality is relaxed.
- *Loop unrolling*: a loop is unrolled when the body of the loop is replicated a number of times u. The benefits of unrolling are multiple: reduction of the loop overhead, increment of the instruction level parallelism, improvement of register, data cache and TLB locality, etc. However, the main disadvantage on loop unrolling is the code expansion.
- *Loop fusion*: joining two different loops in one may increase the register and cache locality since many times it requires less memory accesses and can do a better usage of the registers. This optimization also allows to reduce loop overhead.

- *Variable padding*: conflicts when accessing variables depend on the initial addresses of each variable and the initial addresses of each of their dimensions. By separating two conflicting arrays in the memory space (inter-array padding) or by adding some dummy elements to some array dimensions (intra-array padding), conflict misses can be reduced.
- *Merging arrays*: this technique helps to reduce the number of misses by improving spatial locality. It consists of combining different matrices (that are referenced with the same pattern) into a single compound array. The basic idea is that a single cache block will contain the desired elements, avoiding then the possible interferences among each other.
- C*opying*: this technique is based on adjusting the data layout in cache by copying array tiles to temporary arrays that exhibit better cache behavior.

These are just a some examples of different techniques that can be found in the literature. All of them can be very beneficial is some cases, but at the same time they can degrade the generated code if they are not used properly. Thus, all of them require an in-depth analysis of the code, and in particular the locality properties.

Software prefetching

Software prefetching is another different technique proposed to tolerate memory latency. The main objective of this technique, like hardware prefetching, is to bring data to higher levels of the memory hierarchy (typically the first level cache) before these data are demanded by the processor. so that the accesses can be solved with shorter latency.

In software prefetching, the compiler is the responsible of, following certain criteria, introducing special prefetching instructions in the user code. These instructions will fetch the necessary data so that lately, when the actual load/store instruction is executed, the data can be found in cache. The basic idea is that these kind of instruction will not block the processor on a miss, and then these accesses can be served in parallel with the execution of the following instructions. Then, the basic hardware requirements to support software prefetching are: (1) non-blocking caches, and (2) a prefetch instruction. Non-blocking caches were previously reviewed in this section. A prefetch instruction has three properties:

- It has no target register, since the data is fetched to cache.
- It does not block the processor, and then it can be overlapped with other memory references or computations (this is accomplished with the non-blocking cache).

- It does not provoke exceptions, since software prefetching speculates on certain memory addresses that may not be valid and provoke, for instance, page or protection faults.

The majority of current microprocessors incorporate in their instruction sets operations to perform software prefetching. For instance, the MIPS R10000 [75] has an instruction called *PREF* that can fetch data to the L1 or the L2 caches, and, using a special hint, can determine in which set of the cache the data has to be allocated (since it is 2-way set-associative). The PowerPC620 [19] offers instructions called *TOUCH* that allow to pre-charge data in the cache. The Alpha 21164 [21] allows two modalities of prefetching instructions: *FETCH* (normal prefetching) and *FETCH_M* (permits modifications in some or all blocks to be anticipated). Finally, the HP-PA8000 [47] does not offer any special instruction to prefetch data, but it is done by loading data in the register zero. In case this instruction provokes a trap, then it is executed as a *NOP*.

An alternative to having special prefetch instructions is using longer latencies to schedule memory operations. It consists of scheduling selected load instruction with long latencies so that the consumer can find the data as soon as it is scheduled for execution. In the literature this second alternative is called *binding prefetching* (since the prefetch has a target register), whereas software prefetching using prefetch instructions is called *non-binding prefetching*.

1.3. CONTRIBUTIONS OF THIS THESIS

In this thesis we propose some techniques to improve the performance of the cache by making use of a versatile data locality analysis. The main difference between the techniques proposed in this work and the rest of proposals is the explicit use of data locality information in order to guide the optimization. We claim that the non-homogeneous management of all memory references may be key to performance. With this aim, and with the help of the locality information, each memory instruction will be handled by the compiler in the particular way that best suits the underlying cache organization.

The main contributions of this thesis are listed below:

- We propose and make use of a novel data locality analysis that is performed statically with the help of some simple profiling information. In this analysis, a simple but efficient interference analysis is included, which makes the analysis more accurate.
- The information obtained by the locality analysis is passed to the hardware through some special hints in memory instructions in order to manage some specialized hardware added to the simple cache model.

- We propose a novel explicit management of multi-module caches using the hints previously mentioned to decide, according to locality information, the best way of using these storage modules.
- We propose a software prefetching technique to be used in modulo scheduled loops. This technique is shown to find the best trade-off between processor's compute and stall time.
- Finally, we propose a novel clustered VLIW architecture where the L1 cache is distributed among the different clusters. An algorithm to effectively schedule instructions in this architecture is also proposed.

1.4. ORGANIZATION OF THIS THESIS

The different chapters of this thesis are organized in the following way. In Chapter 2, the data locality analyzers used in the rest of the work are presented. We have used in this thesis two different locality analysis: (i) SPLAT, and (ii) Fast CME. Both analysis have in common that obtain their results statically (that is, without the necessity of simulating the program) with the usage of some simple profiling information. This makes both tools very fast, flexible and accurate. The objective of using two different analysis is also to show that the techniques proposed in the rest of chapters are independent of the locality analysis itself.

In Chapter 3, the SPLAT locality analysis is used to obtain characterize the locality of the SPECfp95 programs, which is later used as a motivation for the different techniques proposed in the rest of the thesis.

In Chapter 4, the first technique that make use of the locality analysis is presented. It consists of a multi-module cache with explicit management based on hints that are set by the compiler. The cache is composed of different modules, each one configured to exploit a particular type of locality. Some hints in the memory instructions indicate the hardware in which module a new fetched has to be allocated. These hints are set by the compiler using the data locality analysis.

In Chapter 5, the second proposal that makes use of the locality analysis is described. This chapter presents a study of the interaction between two different techniques: software pipelining and software prefetching. The first one is a very effective technique proposed to schedule loops with the objective of increasing the ILP. On the other hand, software prefetching, as previously commented, is a technique used to tolerate memory latency. After evaluating different alternatives, a novel algorithm to perform software prefetching in software pipelined loops is proposed. This algorithm takes into account both the shape of the dependence graph and the locality properties of the different memory instructions using the data locality analysis.

In Chapter 6, the last application of the locality analysis is presented. In this chapter the locality analysis will be used to schedule instructions in a proposed clustered VLIW architecture. After developing an effective approach to scheduling instructions ignoring memory effects in a clustered architecture with a distributed register file, an algorithm to schedule instructions in an clustered architecture with a distributed cache is proposed. This algorithm will use the locality analysis (in this case, the Fast CME) to select in which cluster is more beneficial to schedule a memory operation.

Finally, Chapter 7 summarizes the main conclusions of this thesis and outlines the future work.

2

DATA LOCALITY ANALYSIS

Data locality analysis is the process by means of which the intrinsic access pattern of memory instructions and their behavior on a given memory hierarchy are studied. This kind of analysis is commonly used to improve the performance of some locality optimizations performed by the compiler (such us blocking, loop interchange, etc.) as well as to study the locality properties of different programs in order to propose new memory architectures. In this chapter we propose a data locality analysis (called *SPLAT*) that will be used to support some of the techniques proposed in this thesis. Moreover, we review a second data locality analysis (called *FastCMEs*) that, although not proposed in this thesis, is used in one of the proposals.

2.1. INTRODUCTION

Memory penalties are one of the main reasons why computers performance is quite below peak performance for most applications. Understanding the source of the problems is the first step towards devising new hardware organizations and/or new code transformations to overcome them.

The user may be interested in quantifying the memory penalties but this information is not enough in many cases. A more detailed explanation of the different causes for these penalties is sometimes required in order to investigate the appropriate optimization. In order to tune a program, a programmer may be interested in knowing its performance, locating those critical parts where most of the memory penalties are produced, identifying which data structures are responsible for most of the cache misses, etc. Examples of the type of information that the user may be interested in are listed below:

- Classifying the different types of cache misses into the three commonly used categories (compulsory, capacity, conflict) can be important to choose among different types of optimizations. Capacity misses could be best reduced by blocking [31][12]; conflict misses by padding [85]; and compulsory misses by prefetching [9][76], among other possibilities.
- Identifying the parts of the program that are responsible for most penalties may help to reduce the optimization effort by focusing on such cases.
- Conflict misses are the dominant type of misses for many numerical applications. Identifying which data structures are responsible for these conflicts may be required in order to eliminate them by means of padding [85] or copying [99], among other possibilities.
- Quantifying the intrinsic reuse of a program can be used as an upper bound of the locality that can be exploited. This is a useful measure in order to know how far from optimal the current performance is.
- Evaluating the memory performance for a variety of cache architectures for a set of applications can be interesting for the design of an embedded processor with a cache memory customized for a particular workload.
- Including some bits in the memory instructions so that the compiler can provide some hints to the hardware regarding the locality exhibited by each memory instruction is becoming a common practice. For instance, the PA7200 has a bit in order to identify memory instructions with only spatial locality [15]. The PowerPC provides the possibility of identifying instructions that do not exhibit much locality and thus, to bypass the cache for such instructions [97]. Having different cache memories specialized in exploiting different types of locality may be a promising alternative to increase the cache

performance as we will see in Chapter 4. In all these cases, the compiler is responsible for providing the information that is codified in the memory instruction and that will determine during execution the proper action that the hardware must take.

The process of obtaining information of the locality characteristics of a given program is known as *data locality analysis*. This analysis has been performed traditionally either at compile-time or at run-time. The former approach has a low overhead but it is relatively inaccurate since there is much information that the compiler does not know. The latter usually takes the form of a memory hierarchy simulator, which is quite accurate but very slow.

Different approaches to analyze the data locality of programs may be found in the literature. These approaches can be classified into three families of techniques:

- Memory simulation.
- Tools based on hardware-counters provided by some microprocessors. Examples of such microprocessors are the Pentium Pro [81], the UltraSparc [103] or the MIPS R10000 [112].
- Static locality.

The following subsections review some previous work on data locality analysis.

2.1.1. Dynamic Analysis

Any data locality analysis methodology or tool can be assessed through three basic criteria: a) *accuracy*; b) *speed*; and c) *flexibility.* By this later term we mean the amount of different information that the analysis may provide and the possibility to analyze different memory architectures.

Memory simulation techniques are very accurate and flexible in general, but they are very slow. Traditionally memory simulators are based on a trace-driven approach [57][34][74][98][32][67][35][5][72]. They cause a significant slowdown in the execution of the analyzed program, which may be of several orders of magnitude. For instance, the slowdown exhibited by all the simulators surveyed in [105] is in the range of 45-6250. This slowdown is obviously unaffordable for some real applications.

More recently, some innovative methods to perform memory simulation have been proposed with the main objective of reducing the exhibited slowdown. The basic idea behind these methods is to find special cases where a memory reference does not affect the simulated memory state, and avoid or keep very low the overhead in these cases. For instance, if we are interested only in the miss ratio, references that cause a cache hit do not require any processing. The hit detection can be performed by software, as it is the case of MemSpy [70], Fast-cache [64], and Embra [110], or it can be done by hardware, as it is the case of WWT [84]. The slowdown exhibited by these techniques depends on the miss ratio. The lowest slowdown has been reported for WWT, which can be as low as 1.4 for caches with very low miss ratios, but it is greater than 30 or 40 for caches smaller than 32 KBytes [63]. The other three techniques exhibit a slowdown of 2-21, which is still quite high, especially if one considers that the locality analysis usually is a part of an iterative process in which multiple analysis and optimization steps are applied repeatedly. Besides, if the required information is more than just the miss ratio (e.g. type of reuse exhibited or type of misses), it may require to process most or all memory references, which will result in a slowdown similar to that of trace-driven simulation approaches. In other words, these innovative methods trade-off flexibility for speed.

Tools based on hardware counters (e.g. [3]) are fast and accurate but they lack of flexibility, since they can only analyze the memory architecture of the actual microprocessor, and they can just provide a limited set of results which depend on the particular counters provided by a particular machine. Important results like number of conflict misses per each pair of data structures cannot be obtained with current hardware counters, unless they are combined with a type of memory simulator.

2.1.2. Static Analysis

Static analysis techniques (e.g. [100][33]) are fast and flexible, but they can have a low accuracy. This loss of accuracy is caused by the unknown information at compile time. For instance, unknown loop bounds or unknown initial addresses of data structures can be rather detrimental to the accuracy of the results.

The static/dynamic approach taken by the SPLAT tool achieves the best trade-off among the three performance criteria: accuracy, speed and flexibility. It is flexible since the static analysis can track many different information among memory references and different memory architectures can be considered. It is accurate (for numerical programs which are the target of the tool), since the information that is unknown at compile time is provided by a profiling. Finally, it is fast since the profiling information is quite simple and it must be generated just once for multiple analysis of the same program. The slowdown of the tool for the analyzed bechmarks ranges from 0.0 to 0.1[1]. As its main drawback, we should mention that the SPLAT tool is not capable of monitoring multi-process workloads or the operating system kernel.

The main characteristics of the different types of tools are summarized in Table 2.1.

[1] On a SuperSPARC/60 workstation

	Accuracy	Speed	Flexibility
Simulation	Very high	Slow-Moderate	High-Low
Hardware counters	Very high	Fast	Low
Static analysis	Moderate	Fast	High

Table 2.1. Summary of locality analysis tools

2.2. Background Definitions

Before presenting the data locality analysis, we first define some memory related terms that are used all along this chapter. The first definition is related to the terms *reuse* and *locality.* Reuse (also called *intrinsic reuse*) is a measure that is inherent in a given program and depends on neither the order in which instructions are executed nor the cache configuration. A reuse occurs whenever a memory instruction references the same data as a previous instruction (that can either be the same instruction or another one). However, when these instructions are executed, some factors may inhibit the exploitation of this reuse in a given memory hierarchy level (for instance, the limited storage of the cache memory). The amount of reuse that is actually exploited by a given memory hierarchy level is referred to as locality of the program with respect to that memory level.

The different types of reuse/locality used in this chapter are defined in [111]. Temporal reuse occurs when the same memory location is accessed several times. It is called *self-temporal* or *group-temporal* reuse/locality depending on whether it is accessed by the same memory instruction or by different instructions respectively. On the other hand, spatial reuse appears when different nearby memory locations are accessed. It is called *self-spatial* or *group-spatial* reuse/locality depending on whether it is accessed by the same memory instruction or by different instructions respectively. Note that an instruction in a loop nest can have a different type of reuse/locality for each loop on the nest

Finally, the last set of terms that we use in this chapter refers to the different types of cache misses. Misses are traditionally classified into three categories [44]: compulsory, capacity and conflict misses. *Compulsory misses* occur the first time a cache block is accessed. This type of misses are also called cold start misses. On the other hand, both capacity and conflict misses can be considered as replacement misses (in other words, the data was in cache, but when it is actually accessed, it is not). Capacity misses happen because the cache cannot contain all the blocks needed during the execution of a program, whereas conflict misses occur in set-associative caches (including direct-mapped) when too many blocks map to the same set.

For the locality analyzer presented in this chapter, the reuse of each memory instruction is computed following the methodology described in [111]. The results are represented as a vector space that identifies the loops in which reuse is found (each dimension corresponds to a loop). We distinguish between two types of temporal and spatial reuse:

a) *Unitary*: the vector has only one element different from zero, that is, vector (0,...,0,n_i,0,...,0) indicates that this reference has reuse after n_i iterations of loop *i*.

b) *Combined*: the vector has more than one elements different from zero, that is, vector (0,...,0,n_i,n_{i+1},...,n_N) indicates that this reference has reuse after n_i iterations of loop *i*, n_{i+1} iterations of loop *i+1* and so on.

The result of this study is a list of the different reuses exhibited for each reference indicating the loop(s) for which each reuse holds.

2.3. STATIC AND PROFILED LOCALITY ANALYSIS

This section describes the proposed tool for data locality analysis, which is called *SPLAT* (Static-Profiled data Locality Analysis Tool). The locality analysis is performed through some static information computed by the compiler and some dynamic information obtained by a simple profiling (see Figure 2.1).

2.3.1. Compiler and Profiler Phases

The static information is aimed at computing the different types of misses that will happen during the execution. Compulsory misses require to compute the intrinsic reuse of data. Capacity misses require in addition to compute the volume of data referenced by each loop iteration. Finally, conflict misses are identified by computing interferences among data references. All this information is summarized in three files:

- *Reuse file*: for each memory instruction and each loop in which it is enclosed, it stores its type of reuse (unknown, none, self-temporal, self-spatial, group-temporal or group-spatial). If the reuse is spatial it also stores the stride (i.e., the difference between the effective address of two consecutive executions). If the reuse is group-temporal or group spatial, it also contains the *distance*, which is defined as the number of iterations before the reuse takes place.
- *Nest loop file*: this file is intended to represent the loop structure of the program. For each loop it stores its parent, which is defined as the loop that encloses it.

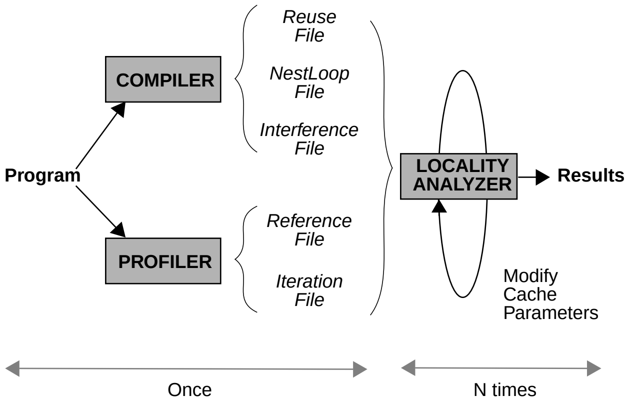


Figure 2.1. Global scheme

- *Interference file*: for each pair of memory instructions (with the same nesting level and without any other loop in between) that have the same reference pattern, it contains their initial addresses if they are known at compile-time[1]. Two instructions have the same reference pattern if their corresponding variables have the same number of dimensions, the size of each dimension is the same in both references, and the expressions that represent the indexing functions for each dimension differ only in a constant value.

The profiling consists of just the number of executions of each basic block, which is a facility provided by many current compilers (e.g., the Sun f77 compiler). From this information, the number of executions of each memory instruction and the average number of iterations of each loop can be derived. These data are stored in the *reference file* and the *iteration file* respectively.

2.3.2. Locality Analyzer

This static and dynamic information is used as an input to the locality analyzer. The locality analysis is divided into three phases: (i) reuse phase, (ii) volume phase, and (iii) interference phase. The first phase identifies all the reuse exhibited by the program. This information is the basis for computing misses. In particular, compulsory misses do not require any additional analysis: they consist of all references without any reuse. The volume phase is targeted to identify capacity misses. Finally, the interference phase computes the conflict misses.

[1] In the SPECfp95 benchmark suite, about 75% of all memory references have their initial address and dimension sizes known at compile-time

```
function qreuse (int i) {
  NN_i[N] = 1;
  ST_i[N] = SS_i[N] = GT_i[N] = GS_i[N] = 0;
  for j=N-1 to 0 do {
    switch (SELFReuse[j]) {
      case NONE:
 A      NN_i[j] = NGIt_j * NN_i[j+1];
        ST_i[j] = TIt_j * ST_i[j+1];
        SS_i[j] = TIt_j * SS_i[j+1];
      break;
      case TEMPORAL:
 B      NN_i[j] = NN_i[j+1];
        ST_i[j] = (TIt_j - 1)* ATIt_j + ST_i[j+1];
        SS_i[j] = TIt_j * SS_i[j+1];
      break;
      case SPATIAL:
        factor = stride / blocksize;
 C      NN_i[j] = (factor * NGIt_j) * NN_i[j+1];
        ST_i[j] = TIt_j * ST_i[j+1];
        SS_i[j] =(factor * TIt_j) * SS_i[j+1] +
                  ((1-factor) * TIt_j) * ATIt_j;
      break;
    }
    GT_i[j] = NGIt_j * GT_i[j+1];
    GS_i[j] = NGIt_j * GS_i[j+1];
    switch (GROUPReuse[j]) {
      case NONE:
      break;
D     case TEMPORAL:
        GT_i[j] += GIt_j * ATIt_j;
      break;
      case SPATIAL:
        GS_i[j] += GIt_j * ATIt_j;
      break;
    }
  }
}
```

Figure 2.2. Algorithm to quantify intrinsic reuse

Reuse phase

In this phase, the different types of reuse exhibited by each reference are quantified. The input to this phase is the *reuse file* that is computed at compile-time following the methodology described by Wolf and Lam in [111].

The quantification of the reuse is performed basically through the function `qreuse(i)` showed in Figure 2.2, which is applied to each memory instruction except for those with unknown reuse[1] (they correspond to references outside loops, or inside loops but with non-linear expressions, or expressions with variables that are not loop indices). The `i` parameter represents the instruction identifier. The analysis starts from the innermost loop and finishes with the outermost loop that includes the instruction `i`, which are denoted by `N-1` and `0` respectively.

[1] References with unknown reuse are assumed to always miss in cache. They represent a 15% of the total number of memory references in the SPECfp95.

The function computes for each particular memory instruction in a particular loop *j* the following values:

- $\mathtt{GIt}_{\mathtt{j}}$: number of iterations with group reuse in loop j.
- $\mathtt{NGIt}_{\mathtt{j}}$: number of iterations without group reuse in loop *j*.
- $\mathtt{TIt}_{\mathtt{j}}$: total number of iterations of loop j.
- $\mathtt{ATIt}_{\mathtt{j}}$: number of executions per each iteration of loop *j*. It is computed as $\prod_{i=j+1}^{N-1} TIt_i$.

The quantification of each type of reuse for each loop in which the reference is enclosed is stored in the vectors `NN` (no reuse), `ST` (self-temporal), `SS` (self-spatial), `GT` (group-temporal) and `GS` (group-spatial). For instance, $\mathtt{ST}_{\mathtt{i}}\mathtt{[j]}$ represents the number of executions of instruction `i` that exhibit self-temporal reuse considering all the iterations of loop *j*. Each type of intrinsic reuse identified by the compiler is quantified as follows (see Figure 2.2):

- *Section A*: the instruction does not have any kind of self reuse in loop *j*. In this case, for each iteration of *j* without group reuse, the number of executions without any reuse is the number of executions without reuse in the loop *j+1* (i.e., $\mathtt{NN}_{\mathtt{i}}\mathtt{[j]=NGIt}_{\mathtt{j}}\mathtt{*NN}_{\mathtt{i}}\mathtt{[j+1]}$). For each iteration of loop *j*, the number of executions with self-temporal or self-spatial reuse is the number of executions with such reuse in loop *j*+1 (i.e., $\mathtt{ST}_{\mathtt{i}}\mathtt{[j]=TIt}_{\mathtt{j}}\mathtt{*ST}_{\mathtt{i}}\mathtt{[j+1]}$).
- *Section B*: the instruction has self-temporal reuse in loop *j*. In this case, the first iteration of loop *j* has the same number of no-reuses as the whole execution of loop *j+1* and the executions corresponding to the remaining iterations reuse the data of the first iteration. Therefore $\mathtt{NN}_{\mathtt{i}}\mathtt{[j]=NN}_{\mathtt{i}}\mathtt{[j+1]}$. Self-temporal reuse is exploited by all executions except for the first iteration. For this iteration, the number of self-temporal reuses corresponds to that exhibited by the next inner loop. Finally, self-spatial reuse is computed as in section A.
- *Section C*: the instruction has self-spatial reuse in loop *j*. In this case, a value called *factor* that represents the percentage of references that access a new cache block is computed. Then, for each iteration of *j* without group reuse that references a new cache line, the number of executions without any reuse is the number of executions without reuse in the loop *j+1*. Self-temporal reuse is computed as in section A. Finally, self-spatial reuse is computed as follows. For those iterations of *j* such that *i* references a new block, the number of self-spatial reuses are the same as those in the next inner loop; and for the remaining iterations, all the executions exhibit self-spatial reuse.

- *Section D*: group reuse is computed as follows (spatial and temporal are treated in the same way). First, for those iterations of *j* such that *i* does not exhibit group reuse, the number of executions with group reuse is the same as that of the next inner loop. For the remaining iterations, all executions exhibit group reuse.

After computing the function `qreuse(i)`, $\texttt{NN}_\texttt{i}\texttt{[0]}$ contains the number of *compulsory misses* of instruction `i`.

Volume phase

A factor that may inhibit the exploitation of reuse is the limited storage of cache memory. In other words, if the amount of different data blocks that are referenced between two consecutive reuses of the same block is higher than the cache capacity (in block units), this reuse cannot be exploited by an LRU fully-associative cache . The resulting cache miss is called a *capacity miss*.

This requires to determine the amount of data that is used by each reference in each loop. This amount of data depends on:

a) *Type of reuse*: calculated in the previous step.

b) *Loop bounds*: obtained from the profiling information.

In this phase, the volume (in cache blocks) that each memory instruction contributes to the total volume of the loops that enclose it is computed. This can be obtained directly from the data computed in the previous phase. For a given loop *j* each execution of instruction *i* that does not exhibit any type of reuse will bring a new block into cache. On the other hand, if a particular execution of an instruction has any type of reuse, it does not bring any additional data into cache. Therefore, the value of $\texttt{NN}_\texttt{i}\texttt{[j]}$ expresses the volume contributed by the instruction *i* to the loop *j*.

Once the volume of every loop has been computed, some reuses are marked as non-exploitable:

- If an instruction has self reuse in loop *j* (either temporal or spatial), but the volume of loop j is greater than the total number of cache blocks, this reuse will likely not be exploited by a conventional cache.
- If an instruction has group reuse (either temporal or spatial) and the volume corresponding to *distance* (see beginning of section 2) iterations of the loop is greater than the total number of cache blocks, this reuse will likely not be exploited either.

Then, the function `qreuse` is computed again but without considering the reuses marked as non-exploitable. The new computed $\mathtt{NN}_{\mathtt{i}}\mathtt{[0]}$, as in the previous phase, represents the cache misses of instruction *i* and the difference with its previous value is the number of *capacity misses* of instruction *i*.

Interference phase

Another factor that influences the locality is the effect of interferences. Typically, interferences or *conflict misses* are defined as those misses that occur in a direct-mapped or *n*-way set-associative cache but not in a fully-associative cache. This kind of misses may have a high impact for cache memories with a low degree of associativity, specially for direct-mapped caches.

The behavior of conflict misses is hard to predict because it depends on various dynamic factors such as memory addresses, instruction order, etc. Interferences may be of two different types: *self-interferences* and *cross-interferences*. Self-interferences occur when different data blocks referenced by the same instruction are mapped onto the same cache location, whereas cross-interferences occur among different memory instructions. The analysis proposed in this section detects a subset of these interferences. The interference analysis is currently implemented for direct-mapped caches. Its extension to set-associative caches is left as future work.

For every array reference and every loop for which it does not exhibit temporal locality, self-interferences are assumed to occur if the following condition is met:

$$cache\_size\_in\_blocks < N * 2^{stride\_family\_in\_blocks}$$

where *N* represents the number of iterations of the loop. The *stride_family_in_blocks* is related to the stride of the reference in the analyzed loop, expressed in cache block units. If the stride is not an integral number of blocks, the stride is rounded up to the next integer. The *stride_family* defined by *x* is the set of strides $\sigma \bullet 2^{x}$ with σ odd [42]. All the strides belonging to the same family (e.g., $12=3 \bullet 2^{2}$ and $20=5 \bullet 2^{2}$ belong to family 2) have the same behavior from the point of view of self-interference.

For each reference and each loop, a *self-conflict ratio* is computed, which denotes the percentage of the *N* iterations of the loop that produce self-interferences. The amount of reuses in outer loops is reduced by this factor due to self-interferences.

Regarding cross-interferences, we focus on what is usually called ping-pong interferences, that is, a pair of instructions that reference different data blocks that map onto the same cache block for every execution. These interferences will inhibit completely the exploitation of any reuse exhibited by the interfering instructions. This type of conflicts is analyzed for each pair of memory instructions that meet the following conditions:

a) Variables whose base address and size of every dimension is statically known. That is, variables allocated at compile-time (75% of all references for the SPECfp95).

b) The difference or “hole” between the addresses (modulo the cache size) of the first element referenced by both instructions is less than the cache block size.

$$hole_{AB} = | \ R_A \bmod cache\_size - R_B \bmod cache\_size \ |$$

c) Both references follow the same pattern (see the above description of the *reference file* for a definition of reference pattern).

For each instruction, a real value between 0 and 1 that represents the *percentage of interference* (*PI*) is defined. If *PI* is 0, this instruction is free of interferences whereas if *PI* is 1, it means that this instruction conflicts with some other instruction for every iteration of the loop. Values in between represent different percentages of interference, that is, the percentage of total iterations in which an instruction misses due to interferences. For two instructions A and B that interfere, this factor is computed as:

$$PI_{\mathrm{AB}} = (block\_size - hole_{AB}) \ / \ block\_size$$

If an instruction conflicts with various other instructions, the maximum *PI* is considered.

The reuse of an instruction *i* in a loop that is not marked as non-exploitable in the volume phase will be exploited only by the percentage of references that are free of interferences, that is, for (1-PI_i) * $nrefs_i$, where $nrefs_i$ is the number of executions of instruction *i*. The rest of references will produce a cache miss.

Then, the function `qreuse` is computed again but considering just the reuses that are free of interferences. The new computed `NN`$_i$`[0]`, as in previous phases, represents the number of cache misses of instruction *i* and the difference with its previous value is the number of *conflict misses*.

2.3.3. Validation

The SPLAT tool estimates the data locality exhibited by a program through some information computed at compile-time and some simple dynamic information obtained by a profiler. The aim of this tool is a fast study of the memory behavior without the necessity of a costly memory simulator. However, this tool would be useless if the obtained results were far from the reality. In this section, we validate the accuracy of the proposed tool by comparing the estimated miss ratios with those obtained through a cache simulator.

Framework

The static analysis used by the SPLAT tool has been implemented using the ICTINEO compiling platform [4]. ICTINEO is a source to source translator that produces a code in which each sentence has a semantics very similar to that of current machine instructions, but the high level information needed for the reuse analysis is retained. Currently, ICTINEO assumes an infinite number of registers and thus, the references produced by spill code are not considered in this work. Optimizations usually applied by current compilers (such as common subexpression elimination, deadcode removal, invariants, etc.) are implemented and are applied to the resulting code. In this way, the resulting code is very similar to the code generated by a production compiler.

For both the tool validation and the application examples of the next section, we have used some programs from the SPECfp95 benchmarks suite. These programs are: *tomcatv*, *swim*, *su2cor*, *hydro2d*, *mgrid*, *applu* and *turb3d*.

A direct-mapped cache has always been considered. The results presented in this section correspond to the profiling/execution of the whole execution of each benchmark using the test input data.

Error in the estimation

In order to validate the tool, the results obtained by simulation and the results produced by the SPLAT tool have been compared. With this goal, we have simulated a direct-mapped cache memory of different capacities (1KB, 8KB and 64KBytes) and various block sizes (16, 32 and 64 bytes). Figure 2.3 shows the results for three programs, two of them showing a high variability in the miss ratio (*tomcatv* and *swim*), whereas the other one has a miss ratio that is much less affected by the cache parameters (*hydro2d*). Besides, *tomcatv* and *swim* are programs with a high conflict miss ratio whereas *hydro2d* has a very low conflict miss ratio.

The first row of graphs shows both the simulated and estimated cache miss ratios for the various configurations of cache. We can see in these graphs that the results obtained by the SPLAT tool are very close to the simulation results. That shows that the tool is accurate for a typical range of cache parameters.

Another way to measure the accuracy of the estimation is to compute the average absolute error per instruction. This error indicates how far from the reality the estimation is for each single instruction. These results are depicted in the second and third rows of graphs.

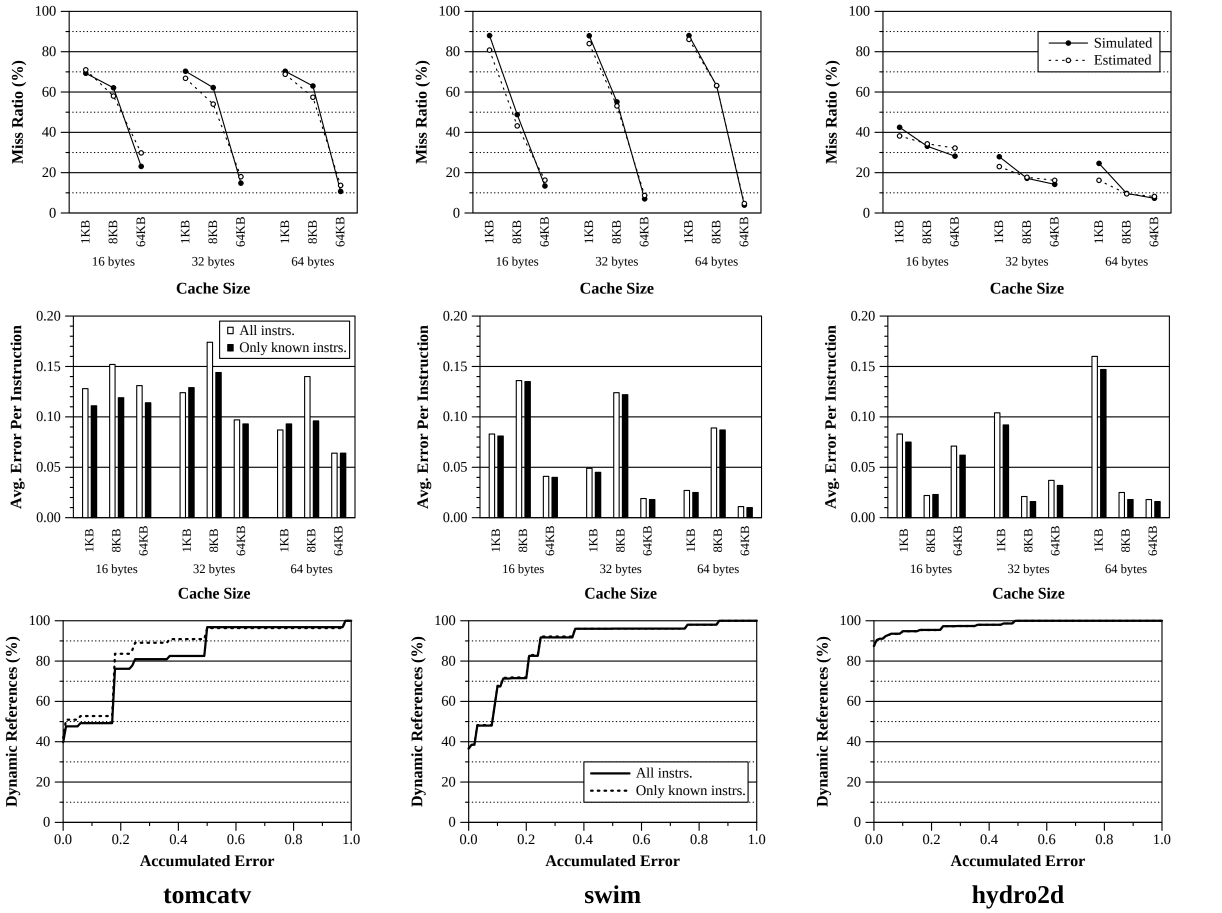


Figure 2.3. Comparison of the tool results against simulation results

The second row shows the average error per instruction, both static (S) and dynamic (D). The static error is computed as:

$$avg\_serror = \frac{\sum_{i}^{\text{NINSTR}} \left| missratio_{est_i} - missratio_{sim_i} \right|}{NINSTR}$$

whereas the dynamic error is computed as:

$$avg\_derror = \frac{\sum_{i}^{\text{NINSTR}} \left| missratio_{est_i} - missratio_{sim_i} \right| \cdot nrefs_i}{\sum_{i}^{\text{NINSTR}} nrefs_i}$$

The black bar represents the error for instructions with reuse known at compile time, whereas the white bar corresponds to the error of all the instructions. The static error is always greater than the dynamic error for all programs and configurations. This means that the analysis estimates better those instructions more frequently executed. Instructions whose reuse is unknown have a high impact on the static error. The impact of these instructions is much lower on the dynamic error, since normally these instructions are out of loops and rarely executed. The average error per dynamic instruction is around or less than 10% for all programs and all the cache configurations.

Finally, on the third row of graphs the estimated error for a particular configuration (in this case, a 8KB cache with 32 bytes blocks) is more detailed. These graphs display the distribution function of the dynamic error per instruction. It can be seen that a very large percentage of dynamic instructions have a very low error. The accuracy of the tool is extremely high for the *hydro2d* program: about 90% of the instructions have no error at all.

2.4. Fast Cache Miss Equations

The second data locality analysis used in this thesis is based on the Cache Miss Equations (CMEs). Although the SPLAT tool previously presented is fast and accurate enough for many programs, it still has some limitations. The two main limitations are that (i) it can only be used to study direct-mapped caches, and (ii) just a subset of all the conflict misses are captured (in particular, only ping-pong interferences).

CMEs allow us to obtain an analytical and precise description of the behavior of any cache memory for loop-oriented codes. Unfortunately, a direct solutions of the CMEs is computationally intractable due to its NP-hard nature. In this section we first review the theory behind the CMEs and then we comment a practical implementation that makes possible to obtain a very accurate and fast locality tool (that we call *FastCMEs*).

2.4.1. Analytical Model

CMEs were originally proposed by Ghosh et al. [33] as an analytical model to describe the behavior of cache memory for a set of memory instructions. The basic idea is to generate, for each memory instruction, a set of constraints (represented through a set of equalities and inequalities) defined over the iteration space of the loop nest in which instructions are enclosed. These constraints represent, for each iteration point in the iteration space, whether the instruction will hit in a given cache or not.

The equations are defined in base of the reuse vectors [111], as used in the SPLAT. Each equation represents a convex polyhedron in R^n, where n depends on the type of equation. For each reuse vector, two kinds of equations are generated:

- Compulsory equations: they represent the first time a memory line is accessed. Misses due to these equations will be counted as compulsory misses.

- Replacement equations: given a reference, these equations represent the interferences with any other reference.[1]

There are two types of compulsory equations: (i) cold miss equations, and (ii) cold miss bounds. The former are those equations that describe the iteration points where a reuse cannot be exploited because it is from an iteration point outside the iteration space. The latter represent the iteration points where the reuse cannot hold since the reference reuses data that is mapped onto a different cache line.

2.4.2. Implementation

The integer points inside each convex polyhedron represent the potential cache misses, and then a direct resolution of the CMEs is based on counting these points. There are several approaches to compute these points. An analytical method would consist of counting the number of points inside the polyhedra generated by the equations. However, this is an NP-hard problem making the problem infeasible due to its huge computing time. A second method to solve the problem could be to traverse the iteration space, that is, all iteration points are tested independently.

In this thesis we have used an analyzer that uses some particular techniques to accelerate the solving of the CMEs. The obtained tool is based on the second methodology previously proposed (traversing the iterations space) and the addition of two optimizations that make the problem feasible:

- Removing empty polyhedra
- Sampling

The first optimization (removing empty polyhedra) was proposed by Bermudo et al. [6] and is based on mathematical techniques. The basic idea is to reduce the number of equations of the system by removing those equations that form empty polyhedra. This is possible using the knowledge that the solutions of our system (points of the iteration space where the reuse cannot be exploited) have to be integer solutions. Then, a polyhedron that only contains real solutions are discarded. This simplification on the complexity of the system result in speeds-up of more than one order of magnitude.

The second optimization (sampling) is based on statistical techniques. As we have mentioned, each iteration point can be tested independently of the others. Based on this property, a small subset of the iteration space could be analyzed, reducing then the computation cost. This optimization (proposed by Vera et

[1] Compared to SPLAT, the CMEs cannot distinguish between replacements due to the limited capacity of the cache (capacity misses) or due to the mapping functions (conflict misses).

al. [107]) uses random sampling in order to select the iteration points to study, and then the miss ratio is computed from them. The use of sampling along with inference theory allows the user to set the desired confidence level and the width of the result interval.

The result is a very fast and accurate tool that can be used to study the locality behavior of any cache for loop-oriented codes. The experiments shown the authors found that, using a confidence of 95% and an interval width of 0.05, the absolute error in the miss ratio is smaller than 0.2 in 65% of the loops from the SPECfp95 and never bigger that 1.0. Moreover, the analysis time required for each program is usually a few seconds and never more than 2.3 minutes.

2.5. Chapter Summary

In this chapter we have presented two data locality analysis techniques that are the base for the different applications and techniques proposed in this thesis.

The first analysis, called *SPLAT*, is proposed in this thesis. It is divided into two steps. The first is performed just once and consists of a basic analysis of some characteristics of the program and a profiling pass (basically, counting number of executions of basic blocks). This first pass produces some information that feeds the locality analyzer. This performs two typical phases that are common to other analyzers, plus an interference phase that can capture a subset of the conflict misses. The overhead of this mechanism is very low since most of the analysis is performed at compile-time and the required profiling support is just a basic block execution count. Besides, we show in this chapter that the proposed mechanism is highly accurate for numeric codes by comparing it with techniques based on simulation. *SPLAT* will be used to study some locality characteristics of the SPECfp95 benchmarks in Chapter 3. Moreover, it is used to support the techniques proposed in Chapters 4 and 5.

The second analysis is called *Fast CMEs* (Cache Miss Equations) consists of a more accurate analysis tool that can be used to study the behavior of any set-associative cache. Although this second analysis is not proposed in this thesis, it will be used in Chapter 6.

An important remark is that although two different analysis have been used in this thesis, the proposed techniques proposed in the following chapters are independent of the locality analysis itself.

3

LOCALITY ANALYSIS OF SPECFP95

An utility of a data locality analysis is the study of the locality behavior of different programs. This study can help researchers to find problems related to memory and devise ways to handle them. This chapter presents differents statistics about the data locality exhibited by the SPECfp95. The tools proposed in Chapter 2 have been used for this study. In this chapter we present studies ranging from the intrinsic reuse of a program (ignoring cache size and associativity) to the actual locality of a program when a real cache configuration is considered. This chapter presents not only quantitative results, but also qualitative numbers that show the reason why a particular reuse cannot be finally exploited in a cache. Quantifying different types of localities as well as different types of cache misses is useful to identify what technique is appropriate to solve (or at least, aliviate) the problem. This kind of study is a previous step before thinking in new schemes to improve memory performance, and is the motivation for the different techniques proposed in the following chapters.

3.1. INTRODUCTION

One of the main goals of the thesis is to show how useful a data locality analysis can be in a compiler. In the kernel of this document we show how this kind of analysis can be used to manage new cache organizations as well as to guide the scheduling of memory instructions in order to reduce the impact of cache misses.

A convenient step before proposing new schemes to improve memory behavior (and, in particular, cache memory) is to study and analyze where the hot spots are and how these problems can be attacked.

This can be done through a data locality analysis. Thus, the first use of a data locality analysis will be the study of the locality exhibited by a set of programs in order to motivate our proposals. In the literature we can find other locality tools as well as papers that reports the kind of studies presented in this chapter (even for the same set of programs). Many of these works are based of the simulation of the programs, and then obtaining a wide variety of results (by modifying, for instance, different cache parameters) is very time costly (as reported in Chapter 2). On the other hand, SPLAT has a low overhead and high accuracy.

3.2. DATA LOCALITY IN THE SPECFP95

This section presents a quantitative analysis of the locality exhibited by the SPECfp95 programs. We provide statistics for seven out of the ten benchmarks. Each program has been compiled with full optimizations and the reported statistics refer to the whole run of them.

We are interested in all types of reuse exhibited by each single memory reference. Consider for instance the following code:

```
DO J=1,M
   DO I=1,N
   ...A(I)...
   ...A(I+1)...
   ...A(I)...
   END DO
END DO
```

Our analysis will conclude that for loop I, the first and third references exhibit group-temporal reuse. Group-temporal reuse is also exhibited by the second and first references (in this case the reuse is after one iteration). Besides, each reference exhibits self-spatial reuse. Now, considering loop J, we have that the three references exhibit self temporal reuse and any pair of references exhibits group-temporal reuse. Assuming that the interference analysis does not detect any interference and that the size of vector A is not higher than the cache capacity, the analysis will conclude that all the reuse can be exploited (the reuse across loop J requires a larger volume that the reuse across loop I, but it is still into the limits of the cache size).

Considering only the last type of reuse in program order as proposed in [72], the analysis would detect only a subset of the different reuses[1]. In particular, it would observe just group spatial reuse for every memory reference. This could suggest that for the above code it is not worthwhile to exploit temporal locality, whereas this is not the case.

3.2.1. Intrinsic Reuse

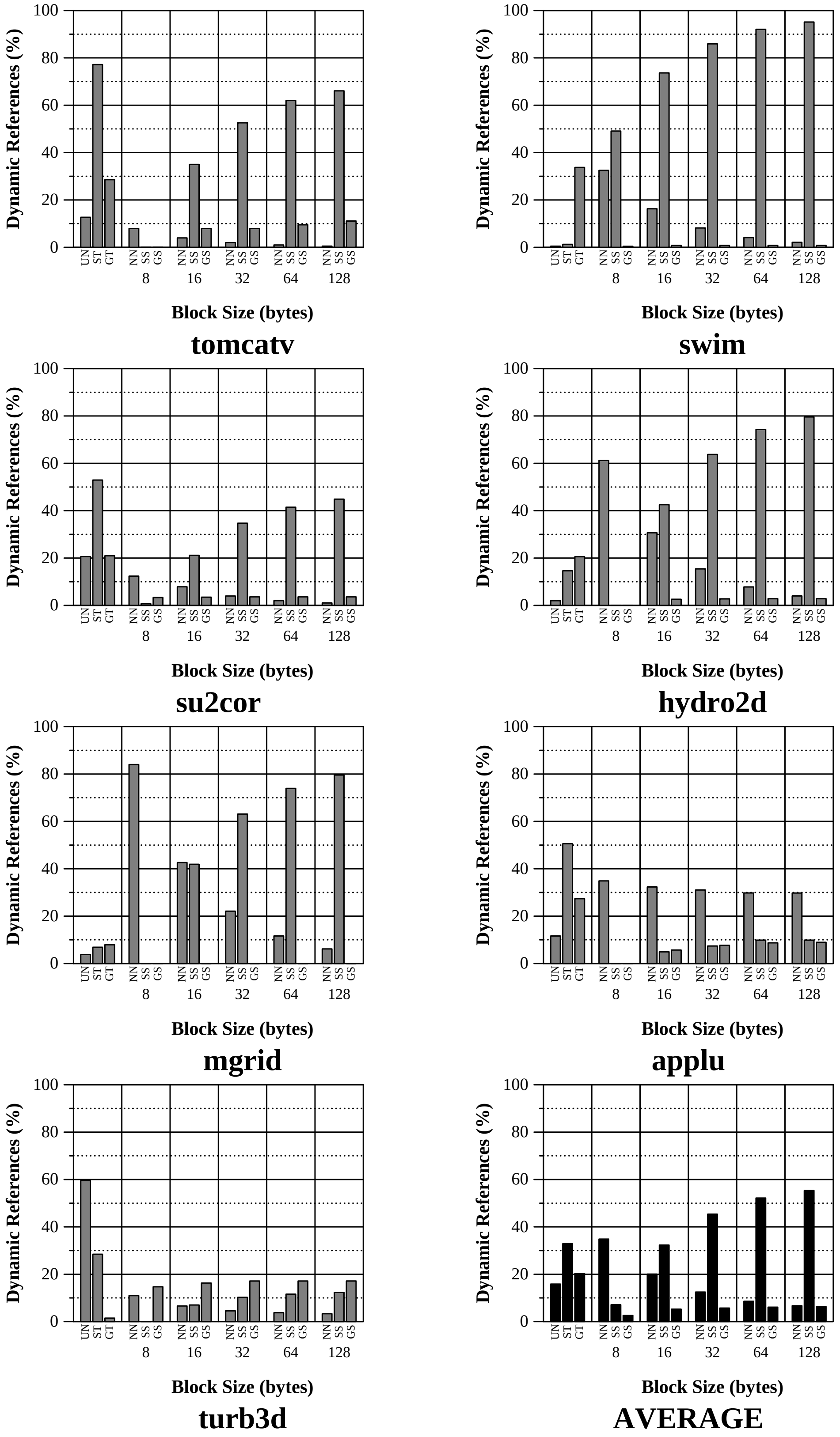
The intrinsic reuse exhibited by a program can be used as a lower bound of the memory bandwidth required by a given program. That is, every reference that does not exhibit any type of reuse will surely require a memory reference to the next level of the memory hierarchy.

Memory reuse can be classified into four categories: self-temporal (ST), self-spatial (SS), group-temporal (GT) and group-spatial (GS). The temporal reuse is independent of the particular cache architecture of the underlying hardware. On the other hand, spatial reuse just depends on the cache line size, in addition to the program characteristics. Regarding group reuse, currently the tool can only analyze the reuse among memory references that are in the same loop, which can result in an underestimation of group reuse. Extending the tool to identify reuse among references in different loops is left for future work.

Figure 3.4 quantifies the amount of reuse of the SPECfp95 for some of the programs and the average. The reuse is quantified for a cache line size ranging from 8 to 128 bytes. In addition to the previously mentioned four categories of reuse, the graphs include a fifth category that corresponds to those references without any type of reuse (NN) and a sixth one that corresponds to those references for which the tool has not been able to detect its type of reuse (UN). Notice that a given reference may exhibit several types of reuse and thus, the different bars may add up to more than 100%.

On average for all programs, it can be seen that self-spatial reuse is the most frequent type of reuse (it is exhibited by 56% of all references). Self-temporal reuse is also significant (33% of references). Group-temporal is the next in importance (20% of references) and finally, group-spatial is the least common one (7% of references). Notice also that group-spatial reuse stabilizes for a 32-byte line size whereas self-spatial reuse provides diminishing returns for a line size greater that 128 bytes.

[1] In [72], what we call reuse is referred as to locality. However, to be consistent with the rest of this paper, we have changed their terminology according to our definition.



tomcatv

swim

su2cor

hydro2d

mgrid

applu

turb3d

AVERAGE

Figure 3.4. Intrinsic reuse

The results for individual programs are very different, and the dominant type/s of reuse depends on the concrete benchmark:

- ***Tomcatv***: the dominant types of reuse are both self-temporal and self-spatial. Group-temporal reuse is also important. In general, the intrinsic reuse for this program is very high (note that the NN bar is very small, even for blocks of 8 bytes).
- ***Swim***: for this program the dominant types of reuse are group-temporal and self-spatial. The other reuses are very low for all block sizes.
- ***Su2cor***: this benchmark presents a high degree of temporal reuse, mainly self-temporal. It is also a program with a high intrinsic reuse.
- ***Hydro2d***: the dominant type of reuse for this program is self-spatial, followed by temporal reuse (both self and group). Finally, group-spatial reuse is almost negligible.
- ***Mgrid***: for this program the temporal reuse is low, both self and group. Likewise, group-spatial reuse is almost null. The dominant type of reuse is self-spatial. Note that the no-reuse bar is very sensitive to the block size.
- ***Applu***: the most relevant aspect of this program regarding its reuse is the low impact of the block size. Note that the increment of spatial reuse or the decrement of no-reuse is almost negligible for blocks bigger than 8 bytes. Besides, this is the program with the lowest amount of reuse, as denoted by the relatively high NN bar.
- ***Turb3d***: in general for this program the reuse is poor, mainly group-temporal and self-spatial reuse. However, note that the percentage on unknown references is very high (about 60%), so the results may not be very accurate for the overall program.

Several conclusions can be drawn from Figure 3.4. First, we can see that in average, self-temporal and self-spatial reuse are the most frequent and none of them is dominant. Group temporal reuse is also quite common whereas group spatial reuse is relatively infrequent. As expected, this results differ from those presented in [72], where it was reported for instance that self-temporal reuse was the least frequent type of reuse[1]. The dominant type of reuse varies significantly for the different benchmarks. Self-temporal is dominant for *tomcatv, applu* and *turb3d*. Self-temporal and group-temporal are the most frequent for *mgrid*. Self-spatial is dominant for *swim, su2cor* and *hydro2d*. Group spatial is always the least common type of reuse. Notice also that in average, the locality analysis can determine the reuse exhibited by about 90% of

[1] Another reason for the discrepancy is the different benchmark suite.

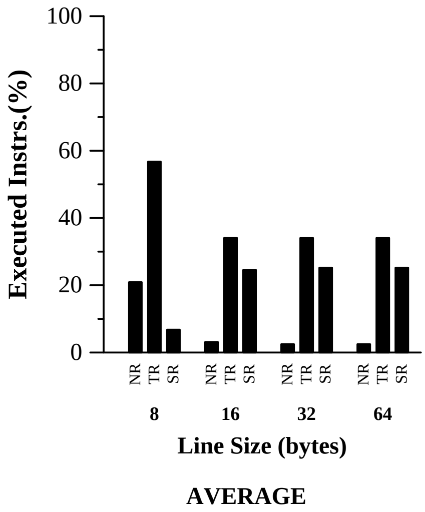


Figure 3.5. Percentage of instructions with just one type of reuse: no reuse (NR), temporal (TR) or spatial (SR).

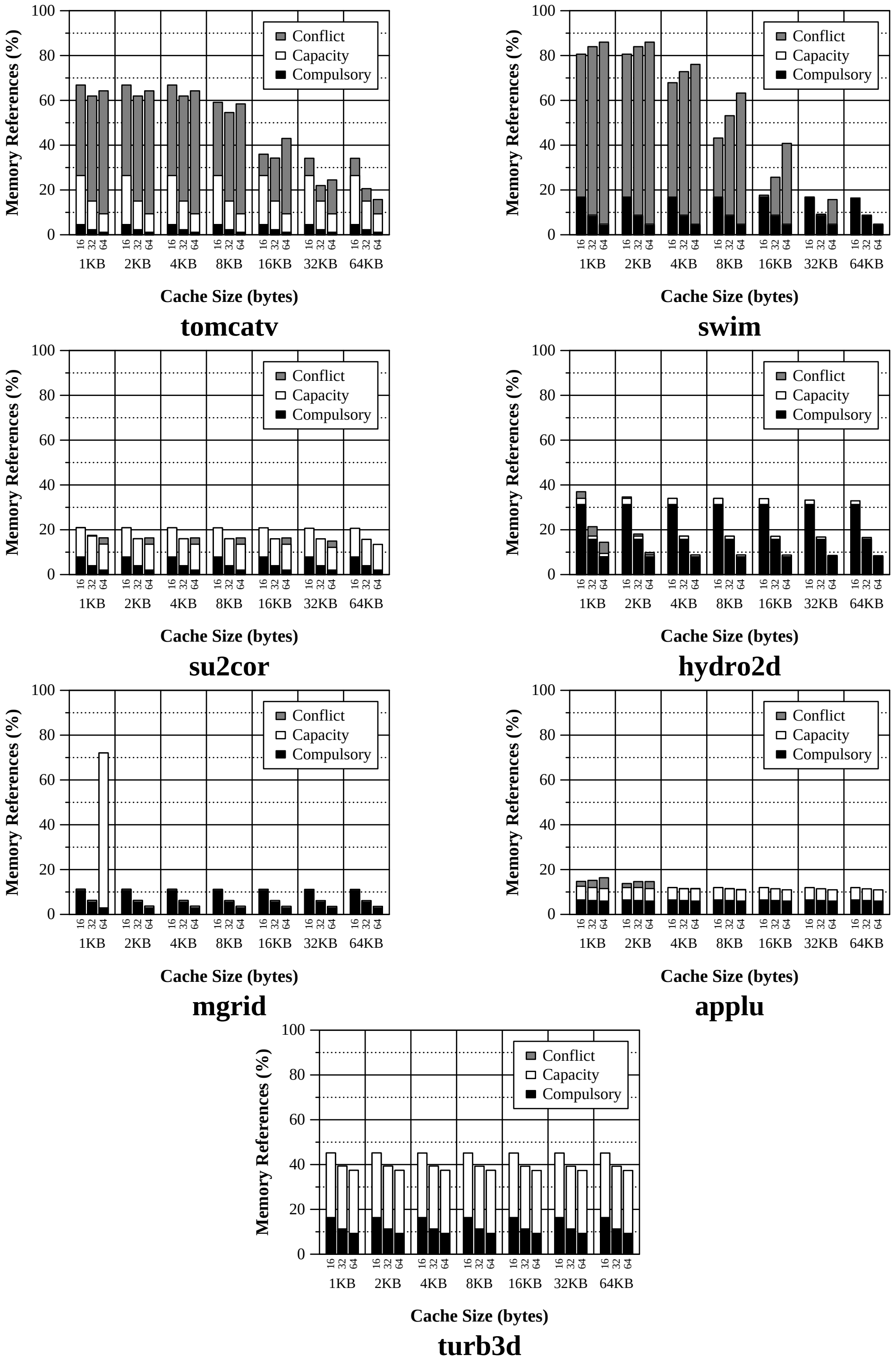
the executed instructions. Finally, it can be observed that almost all the references exhibit some type of reuse.

Figure 3.5 shows the percentage of executed instructions that exhibit just one type of reuse, either spatial or temporal. From now on, the figures present just average statistics over the different benchmarks. From this graph it can be concluded that temporal reuse is the most common type of single reuse, which may suggest the inclusion of a module specialized to exploit temporal locality as it is the case of the dual data cache.

3.2.2. Quantifying Types of Misses

Quantifying the different types of misses may be useful to decide the particular optimization that may best improve the performance of a give program. Misses are traditionally classified into three categories: compulsory, capacity and conflict. Each type of misses can be best reduced with different techniques as underlined in the introduction. Currently the tool can estimate conflict misses just for direct-mapped caches although the extension to set-associative caches is straightforward.

Figure 3.6 shows the miss ratio of the programs studied in this paper for a cache size ranging from 1 KB to 64 KB and a line size of 16, 32 and 64 bytes. For each configuration, the total miss ratio is divided into the three different categories. The y-axis represents the percentage of the total executed memory instructions whose reuse is known (the first column for each graphic in Figure 3.4 - UN column - represents the dynamic percentage of references with unknown reuse).



tomcatv

swim

su2cor

hydro2d

mgrid

applu

turb3d

Figure 3.6. Different kinds of cache misses

For each program, the source of misses may be quite different:

- ***Tomcatv***: this program has a very large number of conflict misses, especially for caches smaller than 16 KB. Increasing the cache size reduces cache conflicts although other techniques like padding could be more cost-effective, as we will show in the next section. Capacity misses are also significant and hardly vary for the considered range of cache capacity. This is because the working set of this program is higher than 64 KB. Finally, note that the most effective line size depends on the cache capacity. For very small caches, the line size has a small effect. For intermediate caches, smaller lines behave better since they significantly reduce the number of conflict misses, due to the larger number of lines. For large caches, the best performance is obtained by the largest line size. This is due to the reduction in capacity misses. Note that increasing the line size may reduce capacity misses although it may seem counterintuitive. This may happen if there are references that exhibit both spatial and temporal reuse but temporal reuse cannot be exploited due to capacity constraints. In this situation, increasing the line size will result in a better exploitation of spatial locality and thus, capacity misses will be reduced. For instance, assume the following code:

```
do i=1,8
   do j=1,1024
     ... A[j] ...
   enddo
enddo
```

  In this example the reference *A[j]* has spatial reuse in loop *j* and temporal reuse in loop *i*. If the cache capacity is 512 elements, the temporal reuse cannot be exploited. Therefore, if the line size is 4 elements, this code will produce 256 (1024/4) compulsory misses (for the first iteration of loop *i*) and 1792 ((1024/4)∗7) capacity misses (the rest of iterations of loop *i*). However, if the line size is 8 elements, there will be 128 (1024/8) compulsory misses and 896 ((1024/8)∗7) capacity misses.

- ***Swim***: the main source of misses for this program is conflict misses. For cache size smaller than16Kbytes, misses due to interferences represent the majority of misses. For a cache of 16Kbytes it is also the dominant source of misses for block sizes of 32 and 64 bytes. Finally, for caches with a size of 32 or 64Kbytes, compulsory misses are the most important cause of miss. For instance, for caches of 64Kbytes, compulsory misses are the only source of misses.

  Note that for this program the effect of capacity misses is almost negligible.

- ***Su2cor***: the low impact of both cache and block sizes in the total number of misses for this program is remarkable. The behavior for all cache sizes is practically constant, and the impact of block size is less than 5%. It can be observed that the major part of misses are due to capacity misses, but the working set of the program is bigger than 64Kbytes, since capacity misses up to that size do not decrease.

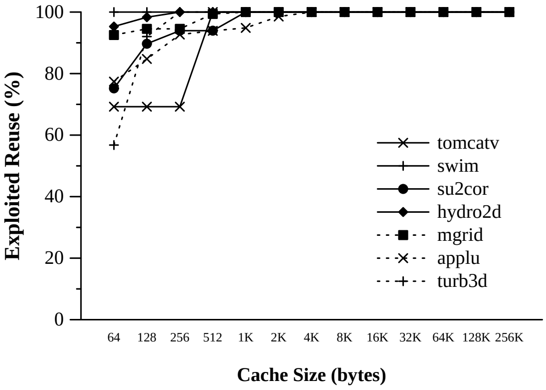


Figure 3.7. Exploiting temporal reuse only

- ***Hydro2d***: for this program all the working set can be stored in a very small cache and thus, increasing cache capacity hardly improves performance. Increasing the line size favors the exploitation of spatial locality and reduces the number of compulsory misses.
- ***Mgrid***: as *hydro2d*, for this program the main cause of misses are compulsory misses. If the cache has just 16 blocks (that is, a 1Kbyte cache with a block size of 64 bytes), there is no space to keep most of the data in cache, so the number of capacity misses is very high.
- ***Applu***: as *su2cor*, the number of misses for this benchmark is almost constant for all configurations.
- ***Turb3d***: this program shows a high miss ratio. However, note that the results are presented for references whose reuse could be studied, and for this program the percentage of unknown references is almost 60% (see Figure 3.4). For this reason, the results of this graph may not be representative of the overall program.

The reuse information together with the quantification of the different types of reuse can be used by the compiler to set appropriately the hints provided by memory instructions in some microprocessors. For instance, if the cache has a bypass capability, those references without any reuse could be marked as non-cacheable. Besides, if two different memory instructions frequently collide, one of them could also be marked as non-cacheable. In this way, the locality exhibited by the other instruction could be exploited, which is better than not exploiting any of both. For instance, in next Chapter will be been reported that this type of analysis when applied to drive a selective caching policy may provide about 25% reduction in average memory access time and 65% reduction in next level memory bandwidth.

Figure 3.7 shows the percentage of temporal reuse that can be exploited with a fully-associative cache that is used only for references that exhibit just temporal reuse (here a fully-associative cache is modelled

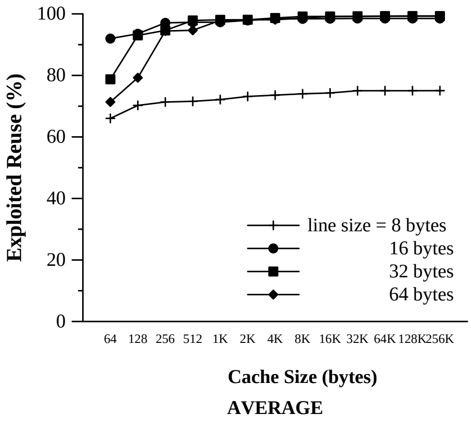


Figure 3.8. Percentage of reuse exploited with a varying cache size without interferences.

by just not considering cache interferences). In this case, the line size is 8 bytes (one double precision float) since a larger line does not make sense because spatial reuse is not present. From this figure we can conclude that a 16 line (128 byte) storage is enough to exploit most of the single temporal reuse. As we have seen in Figure 3.5, these references represent about 35% of the total. Thus, this will be the size of the temporal module of the dual data cache for the experiments of the next chapter.

As pointed out above, a given instruction can have several types of reuse. Given a particular cache organization, we define the percentage of reuse that is exploited as the number of executed instructions that can exploit at least one type of reuse divided by the number of executed instructions that have at least one type of reuse.

Figure 3.8 shows the percentage of reuse that can be exploited for a varying cache size with a line size ranging from 8 to 64 bytes and neglecting the effect of cross-interferences. It can be seen that a cache size of about 1 Kbyte with lines greater than 8 bytes can capture some reuse for practically all the instructions of the analyzed programs with some reuse

Since almost all the references exhibit some type of reuse (as it has been shown in Figure 3.5) and this reuse can be exploited with a relatively small volume, a locality analysis that did not include a interference analysis would incorrectly conclude that it is worthwhile to cache almost all memory references. The percentage of reuse that would be exploited by this approach would be significantly lower than expected due to interferences. This is shown in Figure 3.9 for a varying cache capacity and line size. For instance, comparing the graphs of Figure 3.8 and Figure 3.9 for a 8 Kbytes capacity and 32-byte line size, it can be observed that without interferences nearly 100% of the reuse can be exploited but only 80% of it is actually

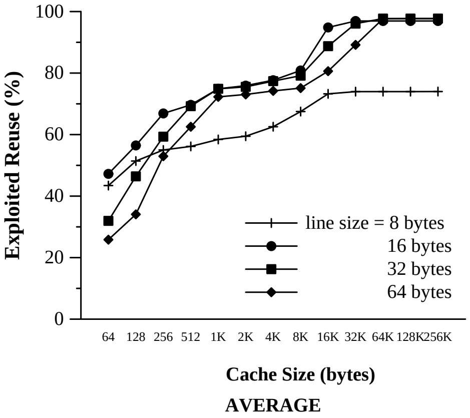


Figure 3.9. Percentage of reuse exploited with a varying cache size considering interferences.

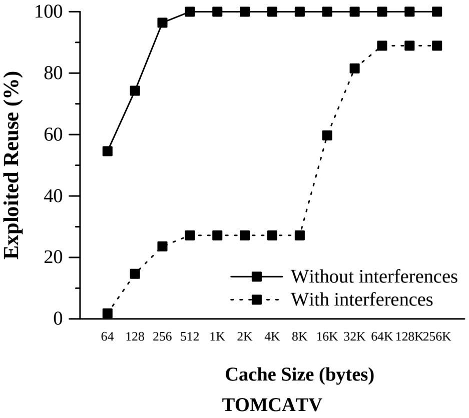


Figure 3.10. Percentage of reuse exploited with a varying cache size with/without interferences for tomcatv.

exploited when considering the effect of interferences. For some programs with a high conflict miss ratio, the effect of interferences is even much more noticeable. This is the case for instance of *tomcatv*. Figure 3.10 compares the percentage of reuse that can be exploited with a varying cache size and a line size of 32 bytes. Whereas 1 Kbyte is enough to exploit all the reuse if there were not interferences, when considering interferences the reuse exploited with a 8 Kbyte cache is just 28%.

3.2.3. Conflicting Data Structures

For those programs with a high percentage of conflict misses, it may be interesting to identify which data structures are responsible for such conflicts. Techniques like padding or copying can be then applied to such data structures to try to reduce these conflicts.

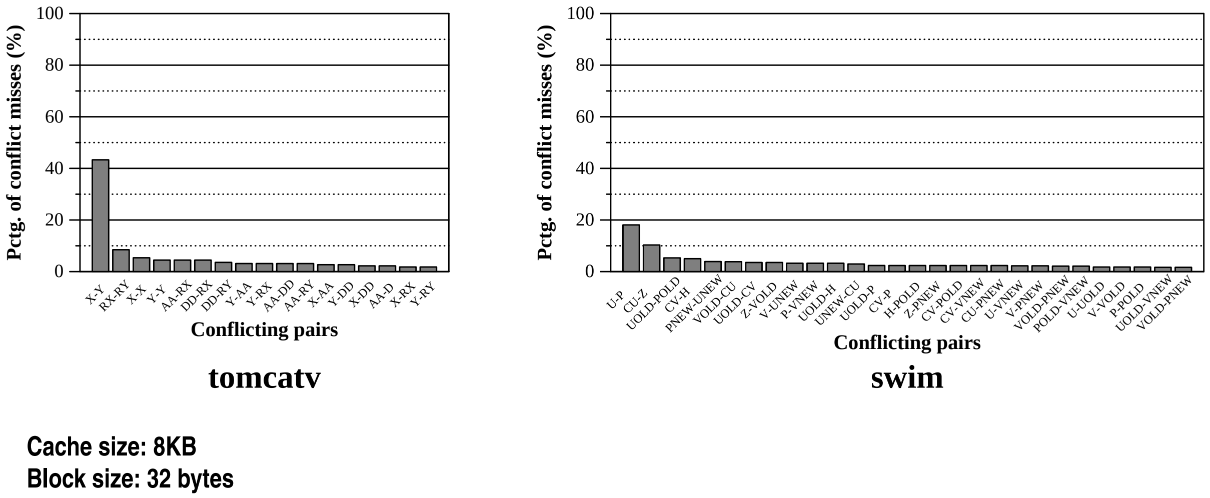


Figure 3.11. Percentage of conflict misses between data structures

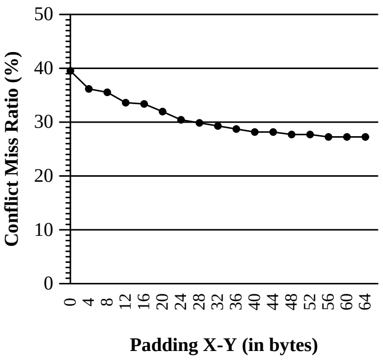


Figure 3.12. Reduction in conflict miss ratio after padding

For instance, Figure 3.11 shows the percentage of conflicts between any pair of data structures, ordered from highest to lowest, for the *tomcatv* and the *swim* benchmarks, for a 8 KB direct-mapped cache with 32 bytes per line. For the *tomcatv* program, it can be seen that data structures X and Y are responsible for the majority of conflict misses. In addition, in Figure 3.11 we can observe that most misses are due to conflicts, which suggest that padding may be an effective technique to reduce memory penalties. For instance, Figure 3.12 shows the resulting conflict miss ratio of *tomcatv* after inserting a number of empty bytes between the two data structures. It can be seen that just with this naive padding scheme, conflict misses are significantly decreased, from 39.5% to 27.2%.

For the *swim* program, conflict misses are more distributed among a larger set of data structures.

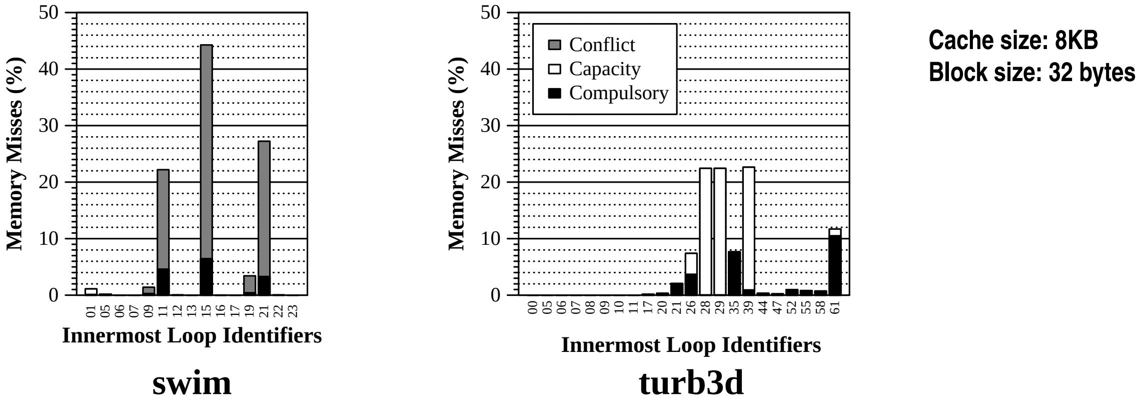


Figure 3.13. Cache misses per innermost loop

3.2.4. Critical Code Sections

Most of the memory penalties are in many cases caused by a very small percentage of the code. Identifying these most penalizing sections may help the programmer/compiler to focus the effort on such parts of the code.

For instance, shows the percentage of cache misses (over the total number of misses) that are caused by every innermost loop, for two applications. Besides, for each loop, its corresponding percentage of misses is split into the three different types: compulsory, capacity and conflict. An 8 KB, direct-mapped cache with 32 bytes per line is assumed.

Note that in both cases the vast majority of misses are due to a very few sections of code: three innermost loops for *swim* and six innermost loops for *turb3d*. For *swim*, most of the misses are due to conflicts whereas in *turb3d*, both capacity and compulsory misses have a significant contribution.

3.3. Chapter Summary

In this chapter we have presented a detailed analysis of the locality exhibited by the SPECfp95 benchmark suite. This detailed evaluation has been performed by means of a new data locality analysis tool that is very fast, which allows to obtain statistics for the whole execution of real programs and many different cache configurations with a negligible slowdown.

We have shown that different programs exhibit very different locality characteristics. Detailed evaluation of the locality exhibited by a program may then be essential to choose the best approach to be taken to improve it.

Fully-automatic optimization tools have proved so far insufficient due to the variety of different scenarios that they should cope with. We then believe that the best approach today towards memory optimization is by means of an iterative (and interactive) process in which repetitive analysis and optimization steps are interleaved until the final result is acceptable. Therefore, the speed of the analysis tool as well as the range of information that it can provide are critical. We have shown that the type of analysis presented in this paper can be very useful for such an approach.

4

LOCALITY SENSITIVE CACHES

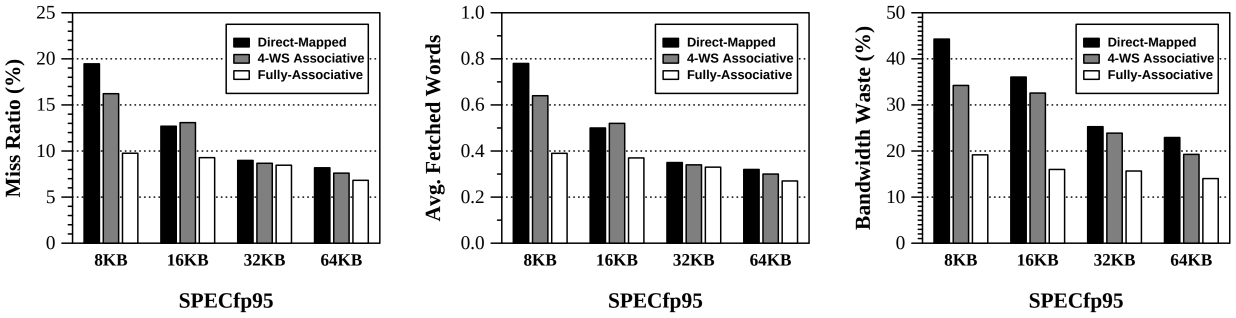
Cache memories are often inefficiently managed, which results in significant memory penalties. An important reason for this poor performance is the homogeneous management of all memory references and the inflexibility of the cache architecture itself, even though different memory references may exhibit a very different locality. Most memory references in numerical codes correspond to array references whose indices are affine functions of surrounding loop indices. These array references follow a regular predictable memory pattern that can be analyzed at compile time. This analysis can provide valuable information like the locality exhibited by each reference, which can be used to implement a more intelligent caching strategy. In this chapter, we present an evolution of different cache organizations that we call locality sensitive caches. This evolution runs from a cache with the capacity of bypassing some data directly to the CPU to a novel data cache architecture composed of different modules, each module exploiting a particular type of locality. The information of which module each fetched data is placed on is passed to the hardware by means of a hint encoded in the memory instructions. This hint is set based on the locality analysis detailed in Chapter 2.

4.1. INTRODUCTION

Due to the great impact that cache performance has on the overall processor performance, current processors assign a large portion of its area to implement a first-level cache (typically split into instruction and data caches). In this way, on-chip caches in modern processors occupy between 1/3 and 3/4 of the total chip area. However, the performance obtained by these caches can still be insufficient for some applications, mainly numeric applications that require large working sets. For instance, Cvetanovic and Bhandarkar reported that the Alpha 21164 is stalled about 50% of the time for the SPECfp92 and the majority of these stalls are due to memory related issues [14].

Increasing the cache capacity/associativity may help but is not necessarily the most cost-effective solution because both capacity and associativity may increase the cycle time. Furthermore, there are several studies (see [45][8] for instance) that show that the cache memory makes an inefficient use of its storage capability. We claim that this inefficiency comes from the uniform management of all memory references. Conventional caches try to exploit spatial reuse by using a *block* (also called *line*) as a transfer unit between the different levels of the hierarchy, and seek to exploit temporal reuse by keeping some recently accessed blocks in the cache memory. All memory references are handled in the same way, that is, they use the same fetch, placement, replacement and write policies [43]. However, this uniform management of all memory references can be very inefficient. In particular, some references can degrade the cache performance by introducing blocks that will not be used in a near future, or blocks where only a small portion of them is used. Such references fetch an unnecessary number of words, wasting memory bandwidth and polluting the cache.

When a reference does not exhibit any type of locality, it results in cache pollution and memory bandwidth waste. The pollution is due to the placement in cache of a non-reusable block whereas the memory bandwidth waste is caused by the additional data brought from L2 cache to L1 cache in the same block as the requested data. To cope with this issue, some current microprocessors provide memory reference instructions that can bypass the cache. On the other hand, when a reference has only temporal locality (i.e., only one data element of each cache block referenced by it is used by itself or any other instruction), it also results in cache pollution and memory bandwidth waste since only one element of the new block will be used. To overcome this problem, a cache could provide a special module to store those data elements with just temporal locality.



(a) Miss ratio **(b)** Avg. fetched words **(c)** Waste of bandwidth

Figure 4.1. Performance of conventional cache architectures averaged for all SPECfp95

For instance, Figure 4.1 shows the results of simulating several conventional cache architectures for some SPECfp95 programs (see Section 4.4.3 for further details about the simulation environment). The graphs show: a) the miss ratio; b) the average number of fetched words (8 bytes) from the next memory level per memory reference (that has a direct relation with the traffic); and c) the percentage of words that are brought into cache but not used before being replaced. Four different capacities are considered: 8KB, 16KB, 32KB and 64KB; as well as three different degrees of associativity: direct-mapped, 4-way set associative and fully-associative. All configurations use a typical line size of 32 bytes. The graphs show the results averaged for all the analyzed programs. We can observe that increasing the cache capacity reduces the miss ratio but the benefits are small beyond 32KB (to obtain a further significant improvement a very large capacity is required). Figure 4.1 also shows that associativity helps but the benefits are more noticeable for small caches. Finally, it can also be observed that for all the configurations there is a high percentage of useless fetched words.

Another important observation is that the spatial locality of each reference may be very different. References with very high spatial locality will benefit from very large cache lines, whereas references with poor spatial locality may favor small cache lines. These differences in spatial locality may be observed if one considers the behavior of different references (or sections) in the same program, or the global behavior of different programs. For instance, Figure 4.2 shows the miss ratio of some SPECfp95 bechmarks (*tomcatv*, *swim* and *su2cor*) for different direct-mapped caches (16KB and 32KB) when the cache line size varies from 16 to 128 bytes. In the graph, light-grey bars show the best line size for each particular program and cache capacity. It can be seen that there are programs that achieve the best miss ratio by using medium or long lines (such as *tomcatv* and *su2cor*). On the other hand, some other programs work better with shorter lines (such as *swim*). This behavior suggests that a unique line size is not the best solution in order to implement a general-purpose cache.

For numerical programs, which are in general more sensitive to the cache performance, the locality of each reference can be estimated quite accurately at compile time using a data locality analysis([31][111][12]). In this chapter, we focus on this type of applications. For other type of applications (non-numerical applications, with a lot of pointers and dynamic structures) a static locality analysis may be unfeasible, but other approaches such as the use of profiling data can be appropriate.

In this chapter we introduce some cache architectures, which we called *locality sensitive caches*. They are composed of one or several modules, each one targeted to exploit a certain type of locality. The selection of where new fetched data is stored is done by some hints in the memory instructions. We propose to use the data locality analysis presented in Chapter 2 to set these hints and manage the proposed caches.

4.2. RELATED WORK

Selective Caching

Selective caching (also called cache bypassing) is a feature of current microprocessors like the PowerPC [97]. It allows some memory accesses that miss on the L1 cache not to allocate any block for the data that is fetched from the next level of the hierarchy. Instead of that, the data is stored just in the target register without polluting the cache with useless data. Some remarkable works for data caches are [17], [1], [36], [53] and [104]. The scheme proposed in [17] is based on a compile-time estimation of data lifetimes. The mechanism proposed in [1] identifies non-cacheable data by means of profiling. The scheme proposed in [36] is based on a run-time managed history table of the most recent load/store instructions. The Cache Bypass Buffers (CBB) are proposed in [53] to reduce interference misses. The approaches proposed in [104] are either hardware-based or make use of simple schemes based on profiling.

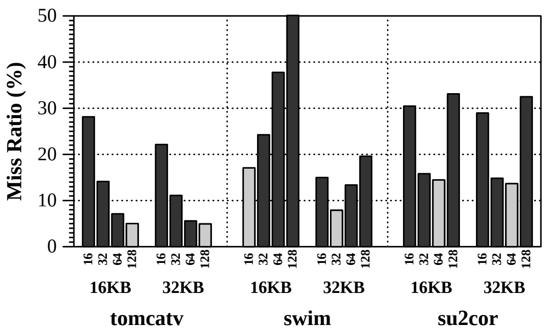


Figure 4.2. Impact of cache line size on total miss ratio for some SPECfp95 benchmarks

Multimodule Caches

Some multi-module cache architectures have previously been proposed in the literature. The *stream buffers* [52] and the *victim cache* [52] where reviewed in the Chapter 1 as classical techniques to improve cache performance. The *stream buffers* perform hardware prefetching in special FIFO queues that are probed in parallel with the data cache. Unlike our proposal, this scheme does not take into account the particular type of reuse exploited by each reference, and thus, unnecessary traffic and low performance can occur for only-temporal and non-strided references. On the other hand, the victim caches has a a primary aim to remove conflict misses by having a fully-associative module where blocks discarded from the main cache are placed. This scheme also makes a uniform management of the cache architecture since all references are handled in the same way. The main drawback of the victim cache is the “blind” swapping management (in the sense that all replaced lines are moved to the victim cache), in addition to the increase in cache port pressure due to the swapping traffic

There are also some works that propose different cache architectures composed of several modules, each one exploiting some particular kind of locality, such as the *dual data cache* [36], the *split temporal/ spatial cache* [73], or the *array cache* [41]. For instance, the dual data cache is composed of two modules, called temporal and spatial. The former is targeted to exploit just temporal locality. The latter is designed to exploit spatial locality, in addition to temporal locality if a reference exhibits both types of locality. In consequence, the temporal module has very short blocks (one 64-bit word is assumed in this study) and the spatial cache has larger blocks (32 bytes per block is assumed here). Using a locality prediction table based on the past history, one of three possible actions in case of a cache miss is selected: a) bring a new long block (32 bytes) and place it in the spatial module; b) bring a new short block (8 bytes) and place it in the temporal module; and c) bring just the requested data, which requires one 64-bit word transaction due to the assumed bus width, and do not place it in any module (that is, bypass the cache).

All these schemes basically attempt to reduce the negative effects of references that exploit only temporal reuse, by just fetching a single word and allocating it in a special module. However, such schemes do not exploit the fact that some references exhibit just spatial reuse.

Rivers and Davidson proposed the *NTS cache* [86]. This architecture dynamically divides cache blocks into two groups: temporal and non-temporal, based on their past reuse behavior. The decision is made through a *detection unit* indexed by effective address. This architecture has a separate small cache (accessed in parallel with the main cache) where non-temporal blocks brought to cache are placed. The basic goal of this scheme is to reduce conflict misses caused by only-spatial references. In [87], it is proposed a modification of the *CNA cache* presented by Tyson et al. [104] that also consists of two cache modules, but in this case the *detection unit* is indexed by program counter.

Another multi-module cache is the one proposed by Johnson and Hwu [51]. This scheme, unlike previous proposals, dynamically divides memory references based on their frequency of reuse. In this case, the detection unit is accessed by effective address, and not by instruction address. Only frequently referenced data are placed on the main cache, whereas the other bypass the main cache and are placed in a small buffer in order to exploit its possible temporal reuse.

Locality Hints

Including some hints in the memory instructions so that the compiler can provide the hardware with relevant information regarding the locality exhibited by each memory instruction is becoming a common practice. For instance, the PA7200 memory instructions have a bit in order to identify references with only spatial locality [15]. In this machine, every memory instruction includes a hint called *spatial locality only* that indicates that the data referenced by that instruction exhibits spatial locality but not temporal locality. The first level of the memory hierarchy of the PA-7200 consists of two modules: the assist cache and the off-chip cache. The former stores all the data referenced by any instruction while the latter stores the data replaced in the assist cache if the spatial locality hint is not set. In consequence, the assist cache is targeted to any type of reference while the off-chip cache is targeted to store the data excepting those with just spatial locality. The PowerPC provides the possibility of identifying memory instructions that exhibit low locality and thus, to bypass the cache for such instructions [97]. In all these cases, the compiler is responsible for providing the information that is encoded in the memory instruction and that will determine during execution the proper action that the hardware must take. A more general approach is taken by the HPL-Playdoh architecture [55] and more recently by the IA-64 architecture [48]. This latter architecture emphasizes the philosophy of passing information from the compiler/profiler to the hardware by making it explicit in the ISA. This information may be related to different issues, such as dependences, speculation, data locality, etc.

Note that an important difference of the schemes proposed in this chapter in comparison to all these previous proposals is the explicit management of the placement of fetched blocks. The information of whether an access bypass the cache, or in which module a new fetched block is stored depends on some hints that are set in the memory instruction at compile time. Therefore, the additional hardware to magage the proposed cache organizations is minimal.

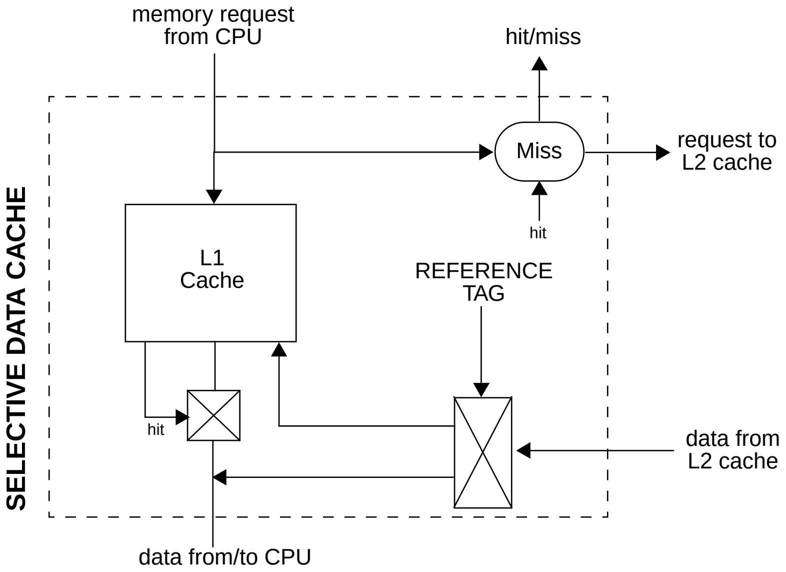


Figure 4.3. Block diagram of the selective data cache

4.3. Selective Data Cache

In this section use the static locality analysis proposed in Chapter 2 to manage the selective data cache organization. As shown in Chapter 3, we know that a substantial percentage of references do not exhibit locality for some programs, in many cases due to cache conflicts. This observation motivates the use of the selective cache managed by software.

4.3.1. Cache Architecture

The selective cache considered in this chapter is like the scheme proposed in [36]. In that case, it was based on a run-time managed history table of the most recent load/store instructions. Figure 4.3 shows the block organization of the selective cache. When a new data is fetched from L2 cache, the selection hardware sets the *reference tag* signal to indicate if the data has to be stored in the L1 cache or just returned to the CPU. In this section, the seletive cache is like a conventional cache in which all the memory instructions have an additional bit that is set by the compiler (corresponds to eh reference tag). In case of a cache miss, this bit controls whether a new block is brought from L2 cache and placed in L1 or just the missing data is requested from L2 and it bypasses L1 cache. We assume a 64-bit data bus between L1 and L2. Thus, this is the bandwidth spent by any bypassing request regardless of the actual size of the required data.

4.3.2. Locality Analysis

The static locality analysis used to manage the selective data cache is based on the analysis proposed in Chapter 2. It consists of the three main steps of that analysis: reuse analysis, interference analysis and volume analysis. We will use the sample code in Figure 4.4 to show how the analysis works and memory instructions are finally tagged with the appropriate locality hint.

We restrict the locality analysis to references inside loops, which represent the majority of references. The locality analysis estimates the type of locality for both scalar and vector references. For the latter, the locality analysis is performed just for array references where the array indices are affine (i.e., linear) functions of surrounding loop indices. In the analyzed benchmarks, the references that were handled by the analysis represent about 90% of the number of dynamic memory instructions. For the remaining references, it is assumed that they exhibit spatial and temporal locality, and then they are tagged to the spatial module.

As in the locality analysis detailed in Chapter 2, the first step of the analysis is the reuse analysis. The result of this phase is a list of the different reuses exhibited for each reference indicating the loop(s) for which each reuse holds. For instance, the reuse analysis of the sample code will produce the following result:

REFERENCE	Reuse in J	Reuse in I
A(J)	self-spatial	N.A.
B(I,J)	no reuse	self-spatial
C(I,J)	no reuse	group-temporal (trailing) with C(I+1,J) and self-spatial
C(I+1,J)	no reuse	self-spatial
D(I,J)	no reuse	self-spatial
E(1,J)	no reuse	self-temporal

```
DO J = 1, 10, 1
  A(J) = 0
  DO I = 1, 1000, 1
    B(I,J) = C(I,J) + C(I+1,J)
    D(I,J) = E(1,J)
  ENDDO
ENDDO
```

Figure 4.4. Sample code.

The main difference between the locality analysis used to manage the selective data cache and the analysis proposed in Chapter 2 is that the order of the volume phase and the interference phase has been interchanged. The reason of this modification is because we try to remove some interferences by using the bypass capability of the selective data cache. Thus, when the analysis detects that two memory instructions suffer from ping-pong interferences, then one of the instructions is marked as non-cacheable (and then, bypasses the cache). In this way, the other instruction will be able to exploit its data locality. As instructions that bypass the cache do not contribute to the volume of the loop, the interference analysis is applied before the volume analysis.

The interference analysis tries to identify groups of memory instructions that will repeatedly produce conflict misses due to interferences among them (we assume in this chapter a direct-mapped organization for the selective and the spatial module of the dual data cache). The analysis consists of the following steps:

1) Using the same analysis as detailed in Chapter 2, detect memory instructions that suffer from ping-pong interferences and build an interference graph for each basic block. Remember that a conflict between two references R_1 and R_2 is assumed if they are mapped onto cache at a distance lower than the block size:

$$\left|R_1 \mathtt{mod}\ \mathrm{CacheSize} - R_2 \mathtt{mod}\ \mathrm{CacheSize}\right| < Block\mathrm{Size}$$

Potential conflicts are analyzed for each pair of references and they are identified in the interference graph by means of an edge (we will later show an example).

2) Remove interferences. If two instructions with some type of reuse interfere, their respective reuse cannot be exploited since the block brought in cache by any of them will be evicted immediately by the other, before it is reused. The objective of this step is to tag some of the interfering instructions as non-cacheable so that the remaining instructions do not interfere and therefore their reuse can be exploited.

The algorithm works as follows: in the interference graph, the node with the maximum number of edges is chosen. This reference is labeled as non-cacheable, and its edges are removed. Then, the process is repeated until the graph has no edges.

If we apply this analysis to the example of Figure 4.4, the results are shown in Figure 4.5. We have supposed that the initial interference graph is the one on the left. The selected reference is `D(I,J)`. Thus,

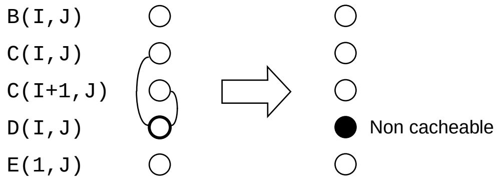


Figure 4.5. Interference analysis for code of Figure 4.4

this reference will be tagged as non-cacheable and it will not be cached despite of having reuse. However, the reuse exhibited by `C(I,J)` and `C(I+1,J)` can be exploited.

The final step of our analysis if the volume analysis as proposed in Chapter 2. If we apply this analysis to our example of Figure 4.4 the results are the following, assuming that the block size is 4 data elements and the cache has 256 blocks:

REFERENCE	Contributed volume to loop I (# of blocks)	Contributed volume to loop J (# of blocks)	
B(I,J)	1	250	
C(I,J)	1	250	
C(I+1,J)	0	0	
D(I,J)	0	0	← Not cached
E(1,J)	1	10	
Total	3	510	
A(J)	-	1	
Total	3	511	

Consequently, only reuse across loop `I` can be exploited. Therefore, the reference `A(J)` will result in repetitive cache misses even though it has spatial reuse.

After the locality analysis is done, each memory instruction is tagged accordingly: references with no reuse are tagged as *bypass*, and the rest as *cacheable* in the selective cache.

4.3.3. Evaluation

Experimental Framework

The cache experiments presented in this setion have been performed using the following SPECfp95 benchmarks [96]: *tomcatv, swim, su2cor, hydro2d, mgrid, applu* and *turb3d*. All of them are written in Fortran language.

The locality analysis has been implemented using the ICTINEO toolset [4]. Optimizations usually applied by current compilers are implemented in ICTINEO and are applied to the resulting code. Memory references are then instrumented according to the locality analysis results, and the trace obtained from the execution of instrumented code feeds a cache simulator of a selective data cache. A conventional cache is also simulated for comparison. The results presented in this chapter correspond to the execution of the first 100 million of memory instructions of each benchmark.

Performance Results

In this section, the performance of the selective data cache is compared against that of a conventional cache. It is assumed that the cache memory is connected to the next level of the memory hierarchy by means of a 8-byte wide bus. The latency of the next memory level is assumed to be 5 cycles plus an extra cycle per 64-bit word. The conventional and selective caches are direct-mapped, write-allocate and copy-back. Cache size is 8 Kbytes and block size is 32 bytes. The spatial module of the dual data cache is like a conventional cache. The temporal module is a very small (16 words) fully-associative buffer. This size has been proved to be sufficient to store practically all memory references that exhibit only temporal locality (see Figure 3.7 in Chapter 3).

Table 4.1 shows the results of the locality analysis applied to the selective cache. The first column indicates the percentage of memory references that are bypassed. The second column lists the hit ratio for the references that are cached. The last column shows the miss ratio of bypass references on a conventional cache, which caches all references. The two last columns provide an evaluation of the locality analysis. An accurate locality analysis should result in a high hit ratio for cached data and in a high miss ratio for non-cached data.

BENCHMARK	%Bypass	%C-Hit	%B-Miss
tomcatv	42.94	71.18	84.37
swim	57.32	89.09	82.06
su2cor	0.06	93.11	0.83
hydro2d	0.05	84.44	69.15
mgrid	0.04	97.19	38.62
applu	1.92	94.51	9.67
turb3d	5.68	96.73	38.71

Table 4.1. Locality results for the selective cache

One can see in Table 4.1 that the hit ratio of cached references is near or above 90% for most programs. On the other hand, the miss ratio of bypassed references on a conventional cache is high excepting some cases in which the percentage of bypass references is very low and therefore the results are not significant (*su2cor*, *mgrid*, *applu* and *turb3d*).

Figure 4.6 shows a comparison among conventional and selective data caches in terms of hit ratio, average memory access time and average number of words fetched from the next memory level per memory reference. These figures are divided in programs with low locality (*tomcatv* and *swim*) and high locality (the others).

We can see that the selective cache provides a significant improvement in the first group of benchmarks. Compared with a conventional cache, they reduce the average memory access time in about 25% and the amount of data fetches in about 65%. Note that this latter benefit may be very effective to reduce memory bandwidth, which is expected to be an important limitation for future microprocessors [8]. In the second group of benchmarks, where the memory behavior on a conventional cache was already good (see Figure 4.6b), the new cache architectures slightly improve the performance except for one benchmark (*applu*) which experiences a small increase in average memory access time.

Selective caching can play an important role to reduce the negative effect of interferences. Applying the interference analysis presented in Section 4.3.2, reuse can be exploited more effectively as it is shown in Figure 4.7. This graph shows the percentage of exploited reuse for different sizes of the selective and conventional (direct-mapped) caches. Results are averaged for all studied benchmarks. From this figure we

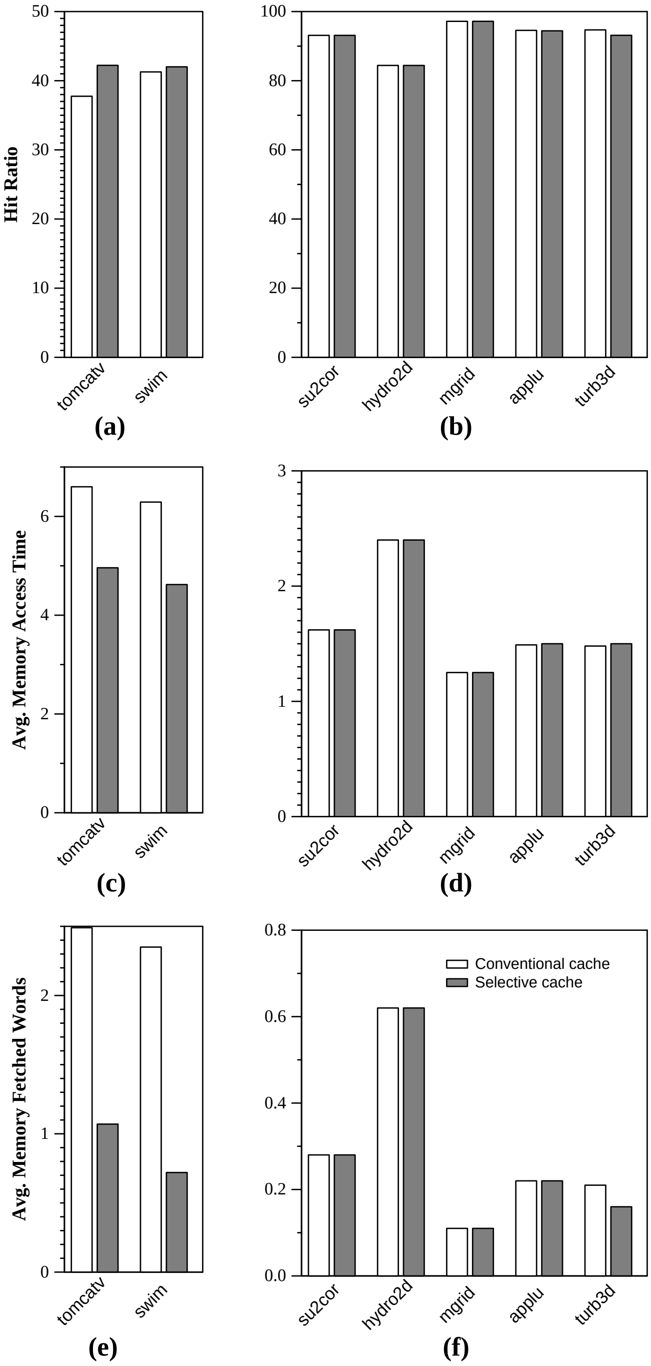


Figure 4.6. Comparison among conventional, selective and dual data caches

can extract, for instance, that a 4 Kbyte selective cache can exploit more reuse than a 8 Kbyte conventional cache. The differences observed are much higher for individual programs with many interferences (*tomcatv* and *swim*).

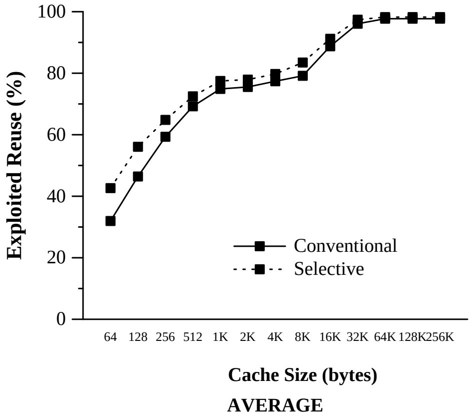


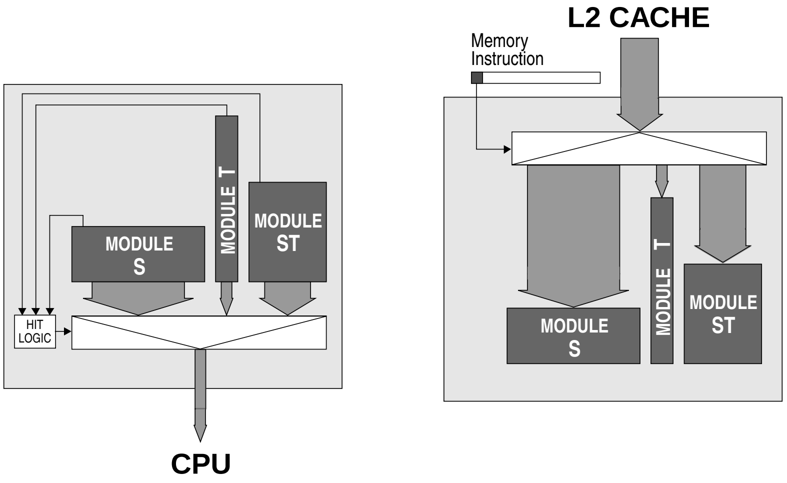
Figure 4.7. Percentage of reuse exploited with a selective cache, varying the cache size and compared with a conventional cache.

The extra bit used to manage the cache do not come free. The most obvious implementation would reduce the range of the constant displacement in memory instructions by a factor of two or four. If the displacement field has 16 bits (which is typical for current architectures) and can be used to address 64KB of data, in the modified instruction set we have that value reduced to 32KB. This may incur in extra instructions if the addressed data is larger. We have measured for the benchmarks considered in this chapter that only 1.32% of dynamic memory instructions executed need extra instructions with 15 bits of displacement (compared with memory instructions that has a displacement of 16 bits), which confirms that the penalty introduced by these additional instructions is negligible.

4.4. MULTIMODULE CACHE

The *LSMCache* (*Locality Sensitive Multimode Cache*) represents an evolution of the dual data cache [36] in the context of locality sensitive caches. It consists of a cache architecture composed of three modules, each one exploiting a particular type of locality. The selection of where the data is placed when it is fetched from the next memory level could be done by a static locality analysis or based on an analysis of some profiling data. As in the selective and dual data caches, the information is passed to the hardware by adding a special field or *hint* to memory instructions. The cache architecture proposed in this work is oriented towards numerical codes, for which module allocation can be completely based on a static locality analysis, due to its high accuracy. However this static analysis is not appropriate for non-numerical codes and a profiled-based analysis may be more effective.

The working of the *LSMCache* is divided into two parts: (1) the compile-time analysis and tagging of memory instructions, and (2) the run-time behavior. Below we first discuss the hardware architecture of the



(a) Cache hit access

(b) Fetching a block from the next level

Figure 4.8. Hardware architecture of the *LSMCache*

LSMCache. Then, the compile-time support, which basically consists of a static locality analysis based in the one proposed in Chapter 2.

4.4.1. Cache Architecture

The hardware architecture of the *LSMCache* is shown in Figure 4.8. The main difference of this architecture with respect to all the previous works on multimodule caches is the clear differentiation of three types of reuse: only-temporal, low-volume self-spatial and the rest (they are later defined). The *LSMCache* is composed of three modules that are referred to as *spatial (S)*, *temporal (T)* and *spatial-temporal (ST)*. Both modules *S* and *T* are small fully-associative buffers, whereas module *ST* is direct-mapped and has larger capacity. The goal of each module is the following:

- *Module S*: it is oriented to exploit *low-volume, self-spatial reuse*. A memory instruction is said to exhibit low-volume, self-spatial reuse if it has self-spatial reuse for a given loop and self-temporal reuse for all inner loops. Intuitively, this is an instruction that has spatial reuse that requires a single cache line to be exploited. For instance, the reference in the next loop:

```
DO I = 1, N, 1
   DO J = 1, M, 1
      ... A(I) ...
   ENDDO
ENDDO
```

has spatial reuse in loop `I`, and temporal reuse in loop `J`. Thus, when a line is fetched and placed in the module *S*, all iterations of loop `J` can take advantage of the temporal reuse without increasing the number of fetched lines. Further iterations of loop `I` will exploit the spatial reuse by accessing other elements of the same line.

Note that either long lines or some simple hardware prefetching technique (prefetching the next or previous block, according to the direction of the stride) may significantly increase the exploitation of the locality exhibited by such references. Both strategies have been evaluated in this work.

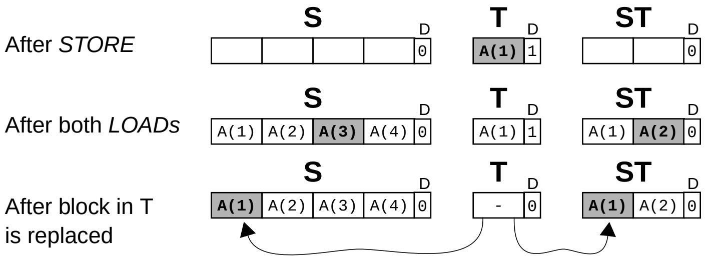
- *Module T*: it is oriented to exploit just temporal locality. In a conventional cache, those references with just temporal locality pollute the cache and waste memory traffic because only one word of the entire line is used. This is avoided by using a special module with short lines to keep these references.
- *Module ST*: this module acts as a conventional cache targeted to exploit both temporal and spatial locality. It stores the data not allocated to the previous modules, that is, data referenced by instructions with both spatial and temporal reuse, such that spatial reuse requires a high number of lines to be exploited. Furthermore, it also stores the data referenced by those instructions whose reuse is unknown. This is the case of references outside loops or references for which the locality analysis cannot determine their locality. Finally, this module also caches those references that cannot be placed in the *S* or *T* modules in spite of meeting the reuse requirement, due to capacity constraints.

Note that due to the different line size, the same data element can reside in several modules at the same time. This may happen when some data is brought to a given module and later on, a reference to a nearby data element brings it again as a part of a larger data block that is placed into a different module. If a copy-back policy is used, the data brought from memory may be stale. Coherency of data is kept in the following way.

For each load instruction, the three modules are checked in parallel. If the data is found in just one module, then it is returned to the processor. In the case that the data is found in more than one module, the data from the module with the smallest line size is returned. Store instructions are also sent to the three modules and those that contain the requested data are updated.

In case of a load/store miss, a new data block is brought into the module indicated by the locality hint included in the instruction. If the replaced line is dirty and it is present in any other module with larger lines, this module is updated. The following simple example can help to understand the coherency methodology:

```
PROGRAM P1          FUNCTION F1(K)    FUNCTION F2(B)
REAL*8 A(N)         REAL*8 K          REAL*8 B(N)
   :                   :                 :
CALL F1(A(1))       STORE K.T         LOAD B(2).ST
CALL F2(A)             :                 :
   :                RETURN            RETURN
LOAD A(3).S         END               END
   :
END
```



In this example each load/store instruction is marked with its hint (*S*, *T* or *ST*), and the **D** bit in a cache line indicates that the line is dirty. First, the store brings A(1) to the module *T*. Then, A(2) is referenced in function F2 and a new block is brought into the module *ST* since it is not in cache, but this block contains a stale copy of A(1). Something similar happens to the following load to A(3), in the main program. The two stale copies of A(1) can reside in cache together with the updated copy, because the data in the smallest cache line will always be chosen. When the dirty line of the module *T* that contains A(1) is replaced, the modules *S* and *ST* are updated.

We will show in Section 4.4.3 that the number of additional accesses required by the coherency mechanism is very low since, on average, only about 0.5% of dynamic memory references hit in more than one module.

4.4.2. Locality Analysis

This section explains the analysis that is performed in order to set the locality hint of every memory instruction for numerical codes. This analysis is divided into three steps:

1) Choose candidate instructions for each module.

2) Sort the candidates in a priority order.

3) Tag each instruction with the appropriate hint.

The selection of the instructions whose referenced data are candidate to be placed in each module is based on a simple reuse analysis. This analysis is very similar to the one used in Chapter 2. The results of this analysis are two vectors that represent the self-reuse and the group-reuse of each memory instruction. The self-reuse is represented by vector ***SRV***. The dimension of this vector represents the nesting level of the memory instruction (that is, the number of loops that enclose the reference). For each loop *i* (loop *0* represents the outermost), the value of ***SRV(i)*** can be ***N*** (no reuse), ***T*** (self-temporal reuse) or ***S*** (self-spatial reuse). The first step of the algorithm is to determine which memory instructions are candidates to be tagged for each module.

The candidates for module *S* are those instructions that meet the following condition:

$$\exists\, i \mid \boldsymbol{SRV}(i) = \boldsymbol{S} \; \underline{\textit{and}} \; \forall\, j > i,\, \boldsymbol{SRV}(j) = \boldsymbol{T}$$

These are those memory instructions that have self-spatial reuse in a loop, and for all their inner loops, if any, they have self-temporal reuse.

The candidates for module *T* are those instructions that meet the following condition:

$$\exists\, i \mid \boldsymbol{SRV}(i) = \boldsymbol{T} \; \underline{\textit{and}} \; \neg\exists\, j \mid \boldsymbol{SRV}(j) = \boldsymbol{S}$$

These are those references that have self-temporal reuse in one or several loops, but do not have self-spatial reuse for any of the loops where they are enclosed.

The rest of the instructions are candidates for module *ST*, including those instructions that are placed outside loops, or for which the reuse analysis cannot be applied.

Note that instructions with group reuse among them exhibit the same self-reuse and thus, they are tagged as candidates for the same module.

The second step, which orders the candidates for the same module according to a priority function, is applied to the *S* and *T* candidates. There are three parameters that determine the order among candidates: (i) placement of the memory instruction in the loop nest; (ii) loop where the reuse has to be exploited; and (iii) stride of the access (only for memory instructions with spatial reuse). The ordering of candidates is based on the following heuristics:

1) The number of reuses for each line brought to the module *S* increases as the nesting level of the loop where spatial reuse is exploited decreases.

2) The volume required to exploit a given type of reuse decreases as the nesting level of the loop that generates the reuse increases.

3) The benefits of spatial reuse are higher as the stride is smaller since the number of reuses for each cache block is higher.

In consequence, memory instructions are first sorted according to their location in the loop nest, from innermost to outermost instructions. For all instructions in the same nesting level, a second step of ordering is applied according to the level of the loop where the reuse is exploited (from outermost to innermost loops). If an instruction can exploit reuse in different loops of the same nest, the innermost of such loops is considered. Finally, instructions placed in the same level of the nest and that exploit their reuse in the same loop level are sorted according to their stride, from smaller to larger. For instance, in the next code:

```
                                  SRVs
     DO I = 1, N, 1
❶       A(I)              (S)
❷       A(I+1)            (S)
        DO J = 1, M, 1
❸          B(J)           (T , S)
❹          C(I)           (S , T)
❺          K1             (T , T)
❻          D(I,J)         (N , N)
        ENDDO
❼       K2                (T)
     ENDDO
```

the allocation of candidates to the different modules and the final ordering of candidates will be as follows:

S Candidates : ❹→❸→❷→❶
T Candidates : ❺→❼
ST Candidates : ❻

The last step of the analysis is the final selection of the tag for each memory instruction. In this step, a volume analysis is applied for both *S* and *T* candidates, in order to determine whether the locality exhibited by each instruction can be exploited given the capacity of the corresponding module. If a reference that was initially candidate for module *S* or *T* does not fit in it, it is finally allocated to module *ST*.

The volume analysis is based on the approach presented in Chapter 2, and it is independently applied to both lists of candidates to modules *S* and *T*. For each reference of the list, from highest to lowest priority, the volume in cache lines that this reference contributes to each loop where it is enclosed is computed and added to the accumulated volume so far. If the accumulated volume of the loop where this reference exploits its spatial/temporal reuse (for modules *S* and *T* respectively) does not exceed the capacity of the module, the memory instruction is tagged with the corresponding module (*S* or *T*). Otherwise, it is allocated to the module *ST* and its contribution to the volume of each loop is subtracted from the accumulated volumes.

For instance, the volume analysis for the *S* candidates in the previous code is the next one:

		Accumulated Volume of Loop `I`	Accumulated Volume of Loop `J`	Condition
❹	`C(I)`	1	1	1 ≤ NLINES ?
❸	`B(J)`	1+M/4	2	2 ≤ NLINES ?
❷	`A(I+1)`	2+M/4	2	2+M/4 ≤ NLINES ?
❶	`A(I)`	2+M/4	2	2+M/4 ≤ NLINES ?

In this example, instruction ***4*** requires just 1 line to exploit its spatial reuse in loop `J`. If this instruction is tagged as *S*, then instruction ***3*** requires 2 cache lines to exploit its reuse. Since both instructions ***4*** and ***3*** have been allocated to module *S*, instruction ***2*** requires 2+M/4 lines in order to exploit its spatial reuse in loop `I`, which is the volume contributed by instructions ***4*** and ***3***, in addition to the line required by itself. Assuming that this volume is still lower than the module *S* capacity, reference ***2*** is tagged as *S*, and then, reference ***1*** is considered. This reference does not contribute any additional volume to any loop, since it reuses the lines referenced by instruction ***2***. Therefore, it is also tagged as *S*.

Finally, the result of the whole locality analysis is reflected in each memory instruction by means of one of the following tags (the stride information is relevant just for those schemes that implement prefetching, as discussed in next section):

HINTS	Module
`00`	*S*, positive stride
`01`	*S*, negative stride
`10`	*T*
`11`	*ST*

4.4.3. Evaluation

Experimental Framework

The previously proposed locality analysis has been implemented in the ICTINEO compiling platform [4]. The programs have been compiled with full optimization (scalar optimizations such as constant propagation, and common subexpressions, deadcode and invariant removal) and the resulting code has been instrumented to generate a trace that feeds a simulator of the cache architecture.

The *LSMCache* has been tested with the following SPECfp95 programs [96]: *tomcatv*, *swim*, *su2cor*, *hydro2d*, *mgrid*, *applu* and *turb3d*. The programs have been executed by using the test input data, and were run for the first 1,500 million of memory instructions (except for programs with fewer instructions).

Schemes without Prefetching

The first experiment evaluates some *LSMCache* architectures that do not incorporate any prefetching scheme. The differences among them are the total size and line size of the module *S*. Table 4.2 summarizes the evaluated configurations. Modules *S* and *T* have both 16 lines and use an LRU replacement. The label *FA* stands for *fully-associative*, whereas *DM* represents *direct-mapped*.

MODEL	MODULE S			MODULE T			MODULE ST			TOTAL CAPACITY (Kb)
	Total Size	Line Size	Assoc	Total Size	Line Size	Assoc	Total Size	Line Size	Assoc	
LSM-S1	512b	32b	FA	128b	8b	FA	4Kb	32b	DM	4.625
LSM-S2	1Kb	64b								5.125

Table 4.2. Basic *LSMCache* configurations

We have compared the proposed architectures against two conventional caches: (a) a 8KB direct-mapped cache (**8KB-DM**), and (b) a 64KB fully-associative cache (**64KB-FA**) (see Section 4.1 in order to compare the results with other conventional cache architectures). The *LSM* architectures are comparable in area and access time[1] to the *8KB-DM* cache. A *64KB-FA* cache requires much more area and a much higher access time that the considered *LSMCache* architectures. This organization is used as a reference point of the miss ratio and memory traffic that could be achieved with a very powerful but also very expensive conventional organization.

Figure 4.9 shows the percentage of dynamic memory instructions tagged as *S*, *T* or *ST* for the *LSM-S1* architecture. We can see that in general, low-volume self-spatial reuse references are the dominant type (these references are allocated to module *S* if there are not volume constraints). However, for some programs, the percentages of references allocated to modules *S* and *ST* are also significant (note that these results are for SPECfp95 programs and, although here the majority of instructions are tagged as *S*, this is not necessary for other codes).

[1] Fully-associative caches larger than the size of the modules *S* and *T* (also with more lines), and with a one-cycle access time have been implemented in commercial processors. An example is the 2KB (64 lines) assist cache of the PA-7200 [15].

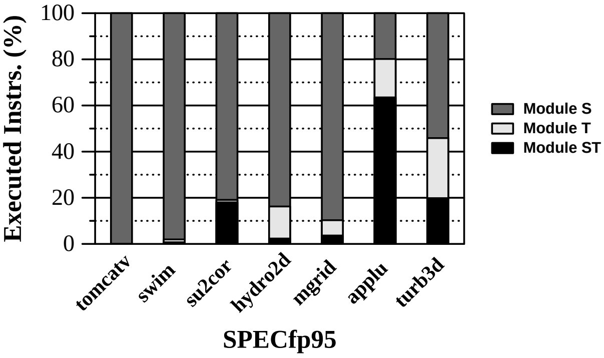


Figure 4.9. Percentage of dynamic memory instructions allocated to each module

Figure 4.10 depicts the miss ratio, average number of fetched words per reference, and percentage of unused words brought into cache, for the *LSMCache* and the two conventional caches. The number of fetched words has a direct relation with the traffic generated between L1 and L2 caches. Moreover, the percentage on unused words (or bandwidth waste) denotes the efficiency of this traffic.

On average both *LSMCache* schemes significantly outperform the *8KB-DM* cache for the three performance figures. For instance, the *8KB-DM* miss ratio is about 2.6 times higher than that of the *LSM-S2* cache. Comparing the proposed schemes, *LSM-S2* performs better than *LSM-S1*, mainly in miss ratio. This tendency is quite uniform for each individual program. Moreover, note that for some programs (*tomcatv*, *swim* and *hydro2d*), *LSM-S2* achieves a better miss ratio than the *64KB-FA* cache even though the number of fetched words is higher. This is due to a better usage of the fetched words. Note also that, on average, the miss ratio of the *LSM-S2* is close to that of the *64KB-FA*, in spite of its much smaller capacity. To achieve this performance, *LSM-S2* requires a higher number of fetched words (which is due to its smaller capacity) but the fetch bandwidth efficiency (which is the reciprocal of bandwidth waste) is comparable to that of the *64KB-FA*.

As explained in section Section 4.4.1, the *LSMCache* requires some coherency operations when a data element resides in more than one module. Note however that this event is rather infrequent. For the *LSM-S1* architecture, this percentage is 0.43% on average for all the programs (the maximum is 2.50% for *applu*, and the minimum is 0.00% for *tomcatv*, *swim* and *su2cor*). Note that in this architecture a datum can be in module *T*, or just in one of the other two since the line size is the same. Regarding the *LSM-S2* architecture, the average percentage is 0.65% (the maximum is 2.11% for *applu*, and the minimum is 0.00% for *tomcatv*).

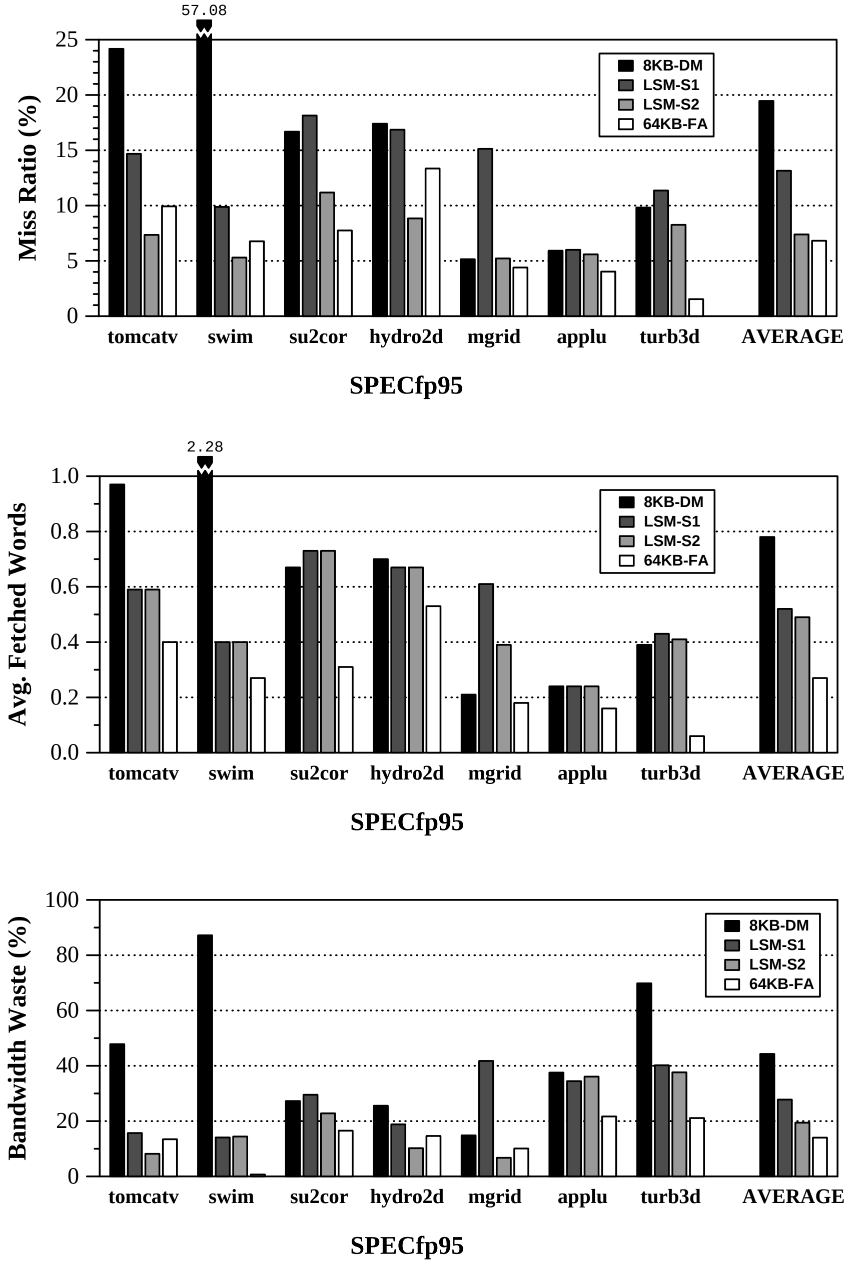


Figure 4.10. Comparison of *LSMCache* without prefetching against two conventional caches

Schemes with Prefetching

Prefetching data is beneficial provided that the cache may anticipate which data will be referenced in the near future. Otherwise, prefetch may harm performance. The detailed characterization of the locality exhibited by each reference allows for an efficient implementation of a prefetch scheme.

We have considered a simple hardware prefetching scheme based on the one-block lookahead schemes (OBL) [94], and extended with a locality analysis. We call the scheme *selective OBL* since the prefetch is performed only for those references that exhibit low-volume, self-spatial reuse (i.e. those allocated to module *S*). Note that this type of locality means that after accessing a data block, it is very likely that the next or the previous block, depending on the direction of the stride, is accessed too. The candidate references for which the prefetch is performed as well as the direction of the stride are provided by the locality analysis described in Section 4.4.2.

Three alternative prefetching schemes, based on those considered in [94], have been implemented:

1) *Always prefetching* (*A*): every time a reference tagged as *S* is performed, a prefetch to the next/previous block is issued.

2) *Prefetching on miss* (*M*): every time a reference tagged as *S* misses (in all modules), a prefetch to the next/previous block is issued as well.

3) *Tagged prefetching* (*T*): every time a reference tagged as *S* accesses a block for the first time since it has been brought to cache, a prefetch to the next/previous block is issued as well.

Note that a prefetch access behaves like an ordinary access, that is, all modules are probed. Table 4.3 summarizes the different *LSMCache* architectures with prefetching that are considered in this section.

MODEL	MODULE S				MODULE T			MODULE ST			TOTAL CAPACITY (Kb)
	Total Size	Line Size	Assoc	Pref.	Total Size	Line Size	Assoc	Total Size	Line Size	Assoc	
LSM-PA	1Kb	32	FA	A	128b	8b	FA	4Kb	32b	DM	5.125
LSM-PM				M							
LSM-PT				T							

Table 4.3. *LSMCache* architectures with prefetching

Both modules *S* and *T* use an LRU replacement. The label *FA* stands for *fully-associative*, whereas *DM* represents *direct-mapped*. Note that the module *S* has 32 lines but the compile-time analysis supposes that it has 16 lines, because some accesses to this module fetch a pair of lines due to prefetching. Thus, the instructions allocated to each module are the same as those in the previous experiments (*LSM-S1* and *LSM-S2*).

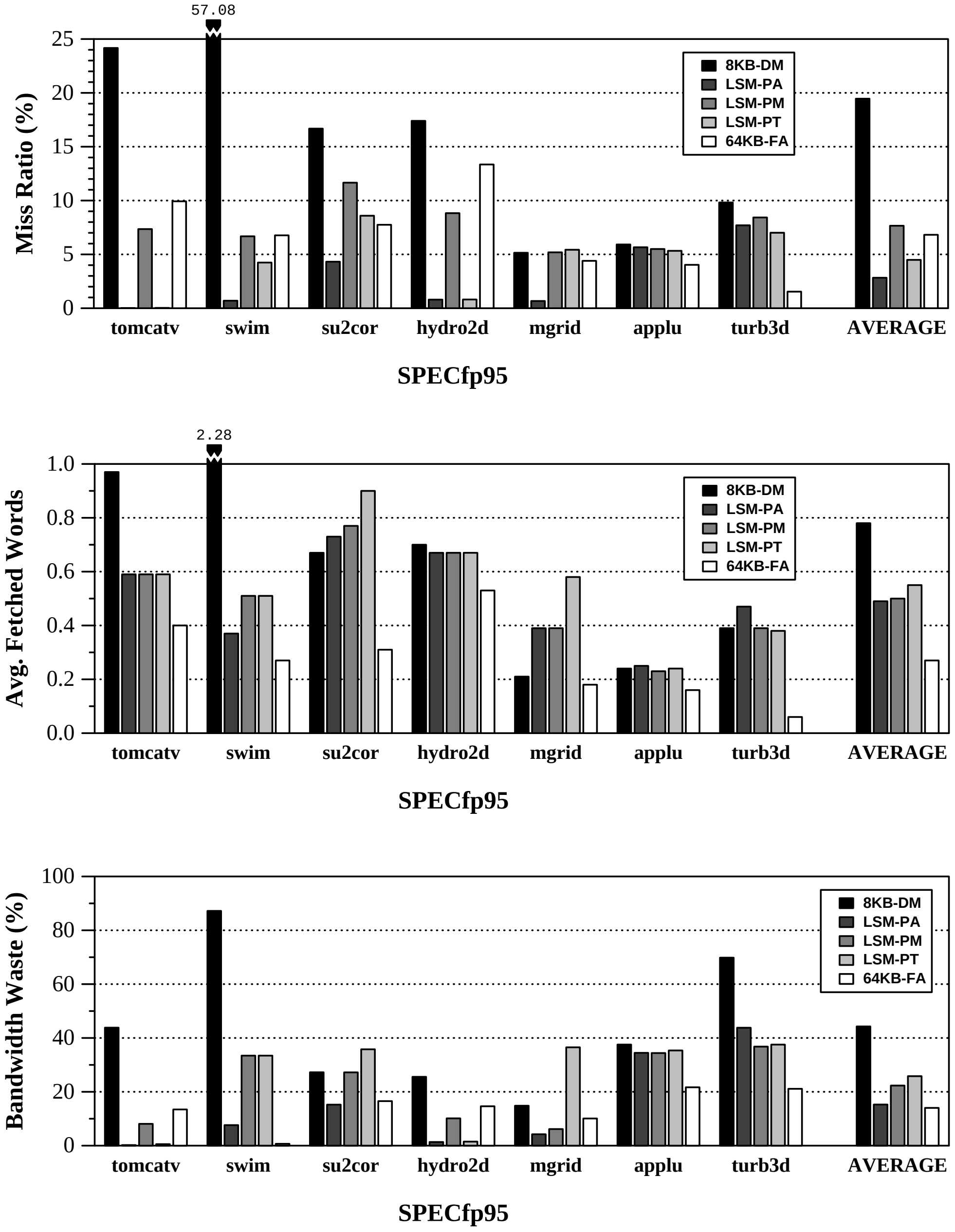


Figure 4.11. Comparison of *LSMCache* schemes with prefetching against two conventional caches

Figure 4.11 depicts the miss ratio, average number of fetched words per reference, and percentage of unused words brought into cache, for the *LSMCache* architectures with prefetching, and compares them with the two conventional caches.

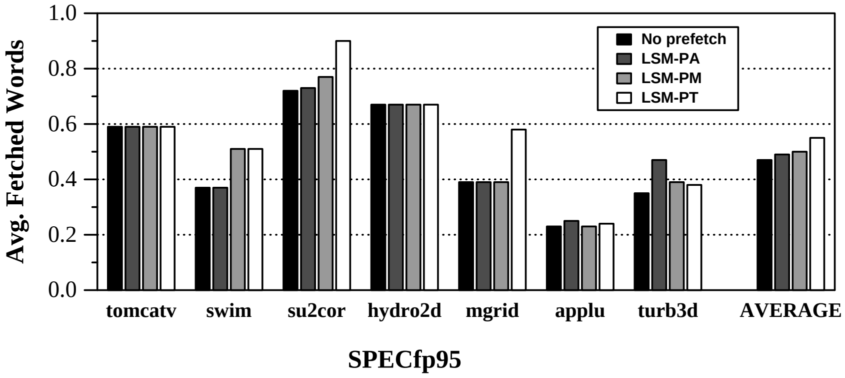


Figure 4.12. Impact of prefetching on fetched words

Regarding miss ratio, *LSMCache* schemes perform much better than the *8KB-DM* cache, achieving an average reduction in miss ratio by a factor of about 7.0 (*LSM-PA*). Note that for some programs the miss ratio is very close to zero. The average reduction in memory traffic is also significant although for two programs (*su2cor,* and *mgrid*) it is somewhat increased. However, the lines brought into cache are better used, as denotes the bandwidth waste graph. Note that these *LSMCache* architectures achieve a lower miss ratio than a fully-associative cache with a capacity twelve times larger (*LSM-PA* has a miss ratio that is 2.5 times lower than that of the *64KB-FA* cache). This requires an increase in the memory traffic by a factor of about two to compensate for the much smaller capacity, but this traffic is efficiently used since the bandwidth waste is of the same order as that of the *64KB-FA* cache.

Among the different *LSMCache* architectures, the best performance is achieved by *LSM-PA*, that is, by the scheme that always prefetches for those data allocated in the *S* module. This scheme generates about the same traffic as the *LSM-PM* (prefetch on miss) scheme but achieves a miss ratio that is 2.7 times lower. The *LSM-PT* (tagged prefetching) scheme has an intermediate miss ratio but generates slightly more traffic. A positive effect of always prefetching is that prefetched data is kept at the top of the LRU stack, and therefore it is usually not evicted before being used. This is an efficient policy since prefetches are very effective, as shown below, because they are driven by a locality analysis.

The effectiveness of prefetching can be evaluated by measuring the additional traffic that they generate. This is shown in Figure 4.12, which depicts the average number of fetched words per reference for the previous configurations and the same cache architectures without incorporating prefetch. It can be seen that the increase in memory traffic due to the prefetch schemes is very low.

As for the schemes without prefetching, the percentage of dynamic references that hit in more than one module has been also obtained, being very low for all three prefetching schemes (0.49% for *LSM-PA*, 0.43% for *LSM-PM*, and 0.35% for *LSM-PT*).

Finally, a drawback of prefetching is an increase in the cache port pressure. Each time a prefetch is issued, all the modules in the cache have to be probed. Table 4.4 shows the average number of cache memory accesses for each dynamic memory reference (without prefetching this number is 1.00):

MODEL	A Always Prefetch	M Prefetch on Miss	T Tagged Prefetch
tomcatv	2.00	1.07	1.22
swim	1.98	1.06	1.08
su2cor	1.81	1.08	1.14
hydro2d	1.84	1.08	1.16
mgrid	1.90	1.05	1.09
applu	1.20	1.00	1.06
turb3d	1.54	1.03	1.23
AVERAGE	1.75	1.05	1.14

Table 4.4. Averaged number of cache accesses per reference

In this table we can see that the *always prefetch* scheme is the one than achieves the lowest miss ratio but at the expense of a higher pressure on cache ports. If this resource is critical, the tagged prefetch scheme may be the best trade-off when both miss ratio and port pressure are considered.

Comparison with Other Multi-Module Caches

We have compared the *LSMCache* with three other multi-module schemes: (a) an 8KB direct-mapped cache with a 512B victim-cache (16 lines of 32 bytes each one)(**8KM-VC**); (b) an 8KB direct-mapped cache with 4 stream-buffers (each one with 4 entries of 32 bytes) with the optimizations proposed by Palacharla et al. [79] (**8KB-SB**); and c) an 8KB 4-way set-associative cache (**8KB-4WA**).

Figure 4.13 compares the miss ratio for a direct-mapped cache (**8KB-DM**), three multi-module caches (**8KB-4WA, 8KB-VC** and **8KB-SB**) and two *LSMCache* schemes (**LSM-S2** and **LSM-PA**), without/with prefetching.

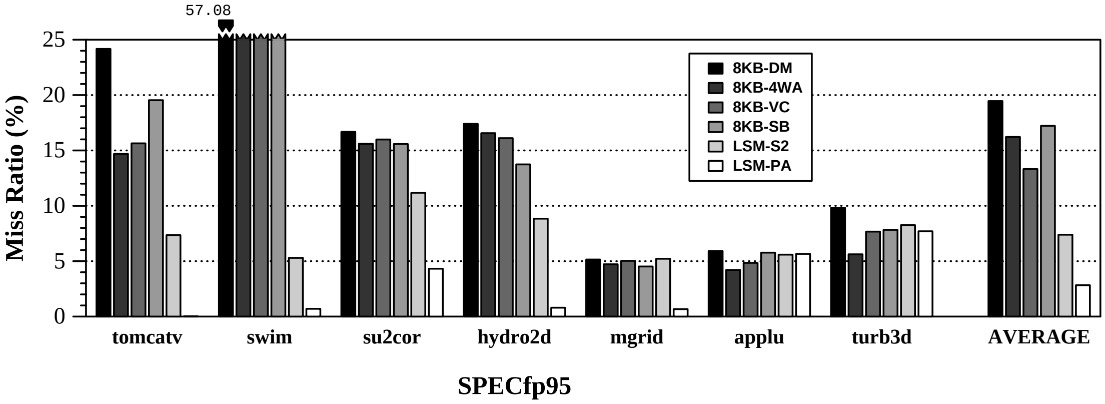


Figure 4.13. Comparison of the *LSMCache* with other multi-module caches

We can see in these graphs that on average *LSMCache* schemes perform better than all the other schemes. Looking at individual benchmarks, only for the last two programs (*applu* and *turb3d*), the *LSMCache* architecture is not the best multi-module scheme.

4.5. CHAPTER SUMMARY

In this chapter we have proposed two new cache architectures. The proposed caches (called *Locality Sensitive caches*) have in common the management through some hints included in memory instructions and that are set at compile time after a locality analysis.

The *Selective Cache* uses these hints to bypass the cache for those memory references without locality. This avoids to pollute the cache with data that will not be reused, and thus makes a better use of the available storage space. Results show that the locality analysis is quite accurate for this type of management. It has been observed that this cache architecture provides a significant reduction in average memory access time and amount of data fetched from the next memory level, specially for programs with a poor locality, when compared with a conventional cache.

The other proposed cache is the *LSMCache*. It is composed of three modules, each module being specialized to exploit a particular type of locality (only-temporal, low-volume self-spatial and the rest). We have observed that for numerical codes the proposed cache architecture eliminates the majority of cache misses with just 5KB of capacity. It has a miss ratio that is about the same as that of a 12 times larger (64KB) fully-associative cache when data prefetching is not incorporated and 2.5 times lower when data prefetching is added (for the most aggressive prefetching scheme). Prefetching is very effective since it is driven by the locality analysis. We have shown that prefetching hardly increases the memory traffic.

The main conclusion of this chapter is that the implementation of smaller caches with a more clever management can be an effective approach to reduce the large area occupied by this component in current microprocessors as well as its access time and power consumption.

5

SOFTWARE PREFETCHING FOR MODULO SCHEDULED LOOPS

This chapter studies the interaction between software prefetching (both binding and nonbinding) and software pipelining for VLIW machines. First, it is shown that evaluating software pipelined schedules without considering memory effects can be rather inaccurate due to stalls caused by dependences with memory instructions (even if a lockup-free cache is considered). It is also shown that the penalty of the stalls is in general higher than the effect of spill code. Second, we show that in general binding schemes are more powerful than nonbinding ones for software pipelined schedules. Finally, the main contribution of this chapter is an heuristic scheme that schedules some memory operations according to the locality estimated at compile time and other attributes of the dependence graph. The proposed scheme is shown to outperform other heuristic approaches since it achieves a better trade-off between compute and stall time than the others.

5.1. INTRODUCTION

Software pipelining is a well-known loop scheduling technique that tries to exploit instruction level parallelism by overlapping several consecutive iterations of the loop and executing them in parallel ([82]).

Different algorithms can be found in the literature for generating software pipelined schedules, but the most popular scheme is called modulo scheduling. The main idea of this scheme is to find a fixed pattern of operations (called kernel or steady state) that consists of operations from distinct iterations. Finding the optimal scheduling for a resource constrained scenario is a NP-complete problem, so practical proposals are based on different heuristic strategies. The key goal of these schemes has been to achieve a high throughput (e.g., [61][49][108][83]), minimize register pressure (e.g., [38][22]) or both (e.g., [46][65][23][66]), but none of them has evaluated the effect of memory. These schemes assume a fixed latency for all memory operations, which usually corresponds to the cache-hit latency.

Lockup-free caches allow the processor not to stall on a cache miss ([60]). However, in a VLIW architecture the processor often stalls afterwards due to true dependences with previous memory operations. The alternative of scheduling all loads using the cache-miss latency increases register pressure and may reduce throughput if recurrences contain memory instructions ([1]).

Software prefetching is an effective technique to tolerate memory latency ([9]). Software prefetching can be performed through two alternative schemes: binding and nonbinding prefetching. The first alternative, also known as early scheduling of memory operations, moves memory instructions away from those instructions that depend on them. The second alternative introduces in the code special instructions, which are called prefetch instructions. These are nonfaulting instructions that perform a cache lookup but do not modify any register.

These alternative prefetching schemes have different drawbacks:

- The binding scheme increases the register pressure because the lifetime of the value produced by the memory operation is stretched. It may also increase the initiation interval due to memory operations that belong to recurrences.
- The nonbinding scheme increases the memory pressure since it increases the number of memory requests, which may produce an increase in the initiation interval. Besides it may produce an increase in the register pressure since the lifetime of the value used to compute the effective address is stretched. A higher register pressure may require additional spill code, which results in additional memory pressure.

In this chapter we investigate the interaction between software prefetching and software pipelining in a VLIW machine. First we show that previous schemes that do not consider the effect of memory penalties produce schedules that are far from the optimal when they are evaluated taking into account a realistic cache memory. We evaluate several heuristics to schedule memory operations and to insert prefetch instructions in a software pipelined schedule. The contributions of stalls and spill code are quantified for each case, showing that stall penalties have a much higher impact on performance than spill code. We then propose an heuristic that tries to trade off both initiation interval and stall time in order to minimize the execution time of a software pipelined loop. Finally, we show that schemes based on binding prefetch are more effective than those based on nonbinding prefetch for software pipelined schedules.

The use of binding and nonbinding prefetching has been previously studied in [58][1] and [9][37][59][76][7] respectively among others. The selective scheduling ([1]) schedules some operations with cache-hit latency and others with cache-miss latency, like the scheme proposed in this paper. However the selective scheduling is based on profiling information whereas our method is based on a static analysis performed at compile time. In addition, the selective scheduling does not consider the interactions with software pipelining. In [20] the authors analyze the effect of modulo scheduling memory operations with either cache-hit latency when they exhibit some type of reuse of cache-miss latency otherwise. Their results show that in average, this scheme is better than the scheme that always uses cache-hit latency but worse than the scheme that always uses cache-miss latency. Our results corroborate this fact. However, the scheme proposed in this chapter outperforms both cache-hit and cache-miss based approaches.

5.2. BACKGROUND ON MODULO SCHEDULING

Modulo scheduling is an instruction scheduling approach for loops [61]. In modulo scheduling techniques all dependences and resource conflicts among operations are considered as the schedule is built. Unlike other techniques that schedule operations from particular iterations with previously scheduled operations from specific iterations, modulo scheduling schedules an operation from all iterations at the same time.

The objective of modulo scheduling is to engineer a schedule for one iteration of the loop such that when the same schedule is repeated at regular intervals no intra- or inter-iteration dependence is violated and no resource usage conflict arises among operations from either the same or distinct iterations. A software pipelined loop via modulo scheduling is characterized basically by two terms: the initiation interval (*II*) and the stage counter (*SC*). The former indicates the number of cycles between the initiation of successive iterations. The latter shows how many iterations are overlapped.

Modulo scheduling algorithms work as follows. A lower bound (*mII*) of the initiation interval (*II*) is computed. This lower bound takes into account the number of available resources of each type and the resource requirements of the loop as well as the dependence constraints. The operations of a single iteration are scheduled in any desired manner but in accordance with the following constraints:

- Input constraints: the inputs of an operation must be generated long enough ago so that they are available at the start of the operation (i.e. dependences must be honored).
- Modulo constraints: Let n_i be the number of resources of type T_i available in the architecture. Then, no more than n_i operations requiring resource T_i may be scheduled for the same time modulo *II*.
  In order to fulfill the modulo constraints, a *modulo reservation table* (MRT) is used. The MRT is a reservation table of length *II* cycles. Each row of the MRT represents all the time slots that are congruent modulo *II* and each column represents a resource. An instruction *u* which is scheduled at cycle C_u is assigned to row C_u mod *II* in the MRT.

Different heuristics to schedule an iteration define different Modulo Scheduling algorithms.

- If no schedule is found when scheduling a single iteration, the *II* is increased in one or several units and the scheduling step is repeated until a correct schedule is found.
- Once a valid schedule for one iteration is found all the implied code motions to form the prolog, the kernel and the epilog are determined by assuming that the same schedule is repeated exactly every *II* cycles.

The execution of a modulo scheduled loop can be divided into three different parts: prolog, kernel (or steady state) and epilog. Figure 5.1 shows these three different stages during the execution of a modulo scheduled loop. In this way, the execution time of the loop can be calculated as:

$$t_{exec} = (NITER + SC\ \text{-}1) \cdot II + t_{stall}$$

For a given architecture and a given scheduler, the first term of the sum (called *compute time* in the rest of the chapter) is fixed and it is determined at compile time. The stall time is mainly due to dependences with previous memory instructions and it depends on the run-time behavior of the program (e.g., miss ratio, outstanding misses, etc.).

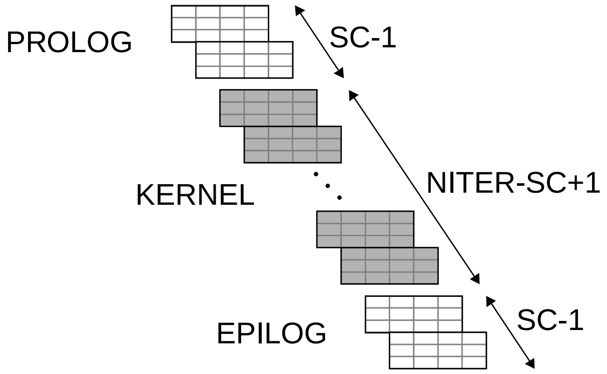


Figure 5.1. Execution stages of a modulo scheduled loop

In order to minimize the execution time, classical methods have tried to minimize the initiation interval with the goal of reducing the fixed part of t_{exec}. The minimum initiation interval is bounded by resources and recurrences:

$$mII = max\ (II_{res}\ ,\ II_{rec})$$

The II_{res} is the lower bound due to resource constraints of the architecture and assuming that all functional units are pipelined, it is calculated as:

$$II_{res} = max\ \left(\forall op \in ARCH, \left\lceil \frac{NOPS(op)}{NFUS(op)} \right\rceil\right)$$

where *NOPS*(*x*) indicates the number of operations of type *x* in the loop body, and *NFUS*(*y*) indicates the number of functional units of type *y* in the architecture.

The II_{rec} is the lower bound due to recurrences in the graph and it is computed as:

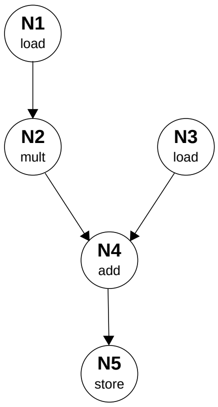
$$II_{rec} = max\ \left(\forall rec \in GRAPH, \left\lceil \frac{LAT(rec)}{DIST(rec)} \right\rceil\right)$$

where *LAT*(*x*) represents the sum of all node latencies in the recurrence *x*, and *DIST*(*y*) represents the sum of all edge distances in the recurrence *y*.

For a particular data flow dependence graph and a given architecture, the resulting *II* is dependent on the latency that the scheduler assigns to each operation. The latency of operations is usually known by the compiler except for memory operations, which have a variable latency. The *II* also depends on the *NOPS*, which is affected by the spill code introduced by the scheduler. The other parameters, *NFUS* and *DIST*, are fixed.

```
DO I = 1, NITER, 1
  A(I) = B(I)*k + C(I)
ENDDO
```

(a) Original code



(b) Data flow dependence graph

	ALU	MEM
0		N1
1	N2	
2		
3		
4		N3
5	N4	
6		
7		
8		N5

Instruction latencies:
load/store : 1-10 cycles
add : 2 cycles
mult : 4 cycles

(c) Code scheduling

	ALU	MEM
0		$N1_0$
1	$N2_0$	$N3_1$
2	$N4_1$	$N5_2$

(d) Kernel

Figure 5.2. A sample scheduling

5.3. ADDING SOFTWARE PREFETCHING

5.3.1. Motivating Example

Conventional modulo scheduling proposals use a fixed latency (usually the cache-hit time) to schedule memory instructions. Scheduling instructions with its minimum latency minimize the register pressure, and thus, reduces the spill code. On the other hand, this minimum latency scheduling can increase the stall time because of data dependences. In particular, if an operation needs a data that has been loaded in a previous instruction but the memory access has not finished yet, the processor stalls until the data is available.

Figure 5.2 shows a sample scheduling for a data dependence graph and a given architecture. In this case, memory instructions are scheduled with cache-hit latency. If the stall time is ignored, as it is usual in studies dealing with software pipelining techniques, the expected optimistic execution time will be (suppose *NITER* is huge):

$$t_{exec_{opt}} = (NITER + 2) \cdot 3 = 3 \cdot NITER + 6 \cong 3 \cdot NITER$$

Obviously this is an optimistic estimation of the actual execution time, which can be rather inaccurate. For instance, suppose that the miss ratio of the N1 load operation is 0.25 (e.g., it has stride 1 and there are 4 elements per cache line). Every cache miss the processor stalls some cycles (called *penalty*). The penalty for a particular memory instruction depends on the hit latency, the miss latency and the distance in the scheduling between the memory operation and the first instruction that uses the data produced by the memory instruction. For the dependence between N1 and N2 the penalty is 9 cycles, so the stall time assuming that the remaining dependences do not produce any penalty is:

$$t_{stall} = NITER \cdot (10 - 1) \cdot 0.25 = 2.25 \cdot NITER$$

and therefore

$$t_{exec} = 5.25 \cdot NITER = 1.75 \cdot t_{exec_{opt}}$$

In this case, the actual execution time is near twice the optimistic execution time. If we assume a miss ratio of 1 instead of 0.25, the discrepancy between the optimistic and the actual execution time is even higher. In this case, the stall time is:

$$t_{stall} = NITER \cdot (10 - 1) \cdot 1 = 9 \cdot NITER$$

and therefore

$$t_{exec} = 12 \cdot NITER = 4 \cdot t_{exec_{opt}}$$

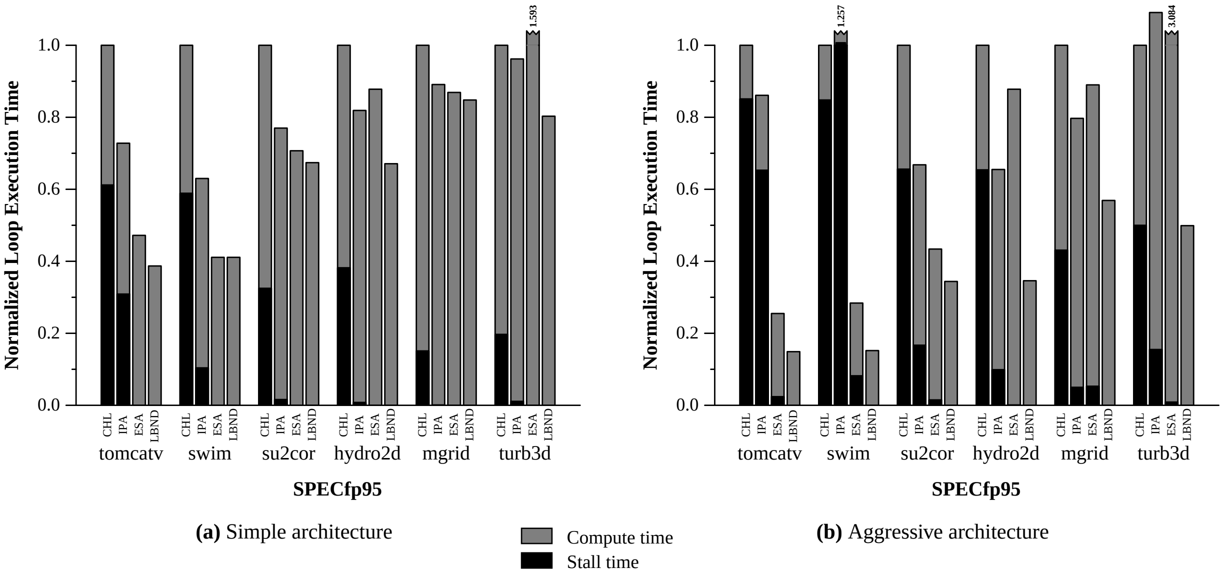
If all memory references were considered, the effect of the stall time could be greater, and the discrepancy between the optimistic estimation usually utilized to evaluate the performance of software pipelined schedulers and the actual performance could be much higher. We can also conclude that scheduling schemes that try to minimize the stall time may provide a significant advantage.

In this paper, the proposed scheduler is evaluated and compared with others using the t_{exec} metric. This requires to consider the run-time behavior of individual memory references, which requires the simulation of the memory system.

5.3.2. Basic Schemes to Schedule Memory Operations

In this section we evaluate the performance of basic schemes to schedule memory operations and point out the drawbacks of them, which motivates the new approach proposed in the next section.

We have already mentioned in the previous section that modulo scheduling schemes usually schedule memory operations using the cache-hit latency. This scheme will be called *cache-hit latency* (CHL). This scheme is expected to produce a significant amount of processor stalls as suggested in the previous section.



(a) Simple architecture

(b) Aggressive architecture

Figure 5.3. Basic schemes performance

An approach to reduce the processor stall is to insert a prefetch instruction for every memory operation. Such instructions are scheduled at a distance equal to the cache-miss latency from the actual memory references. This scheme will be called *insert prefetch always* (IPA). However, this scheme may result in an increase in the number of operations (due to prefetch instructions but also to some additional spill code) and therefore, it may require an *II* higher than the previous approaches.

Finally, an alternative approach is to schedule all memory operations using the cache-miss latency. This scheme will be called *early scheduling always* (ESA). This scheme prefetches data without requiring additional instructions but it may result in an increase in the *II* when memory instructions are in recurrences. Besides, it may also require additional spill code.

Figure 5.3 compares the performance of the above three schemes for some SPECfp95 benchmarks and two different architectures (details about the evaluation methodology and the architecture are given in Section 5.4). Each column is split into compute and stall time. In this figure it is also shown a lower bound on the execution time (LBND). This lower bound corresponds to the execution of programs when memory operations are scheduled using the cache-hit latency (which minimizes the spill code) but assuming that they always hit in cache (which results in null stall time). This lower bound was defined as the optimistic execution time in Section 5.3.1.

The main conclusion that can be drawn from Figure 5.3 is that the performance of the three realistic schemes is far away from the lower bound in general. The CHL scheme results in a significant percentage of stall time (for the aggressive architecture the stall time represents more than 50% of the execution time for most programs). The IPA scheme reduces significantly the stall time but not completely. This is due to the fact that some programs (especially *tomcatv* and *swim*) have cache interfering instructions at a very short distance and therefore, the prefetches are not always effective because they may collide and replace some data before being used. Besides, the IPA scheme results in a significant increase in the compute time for some programs (e.g., *hydro2d* and *turb3d* among others). The ESA scheme practically eliminates all the stall time. The remaining stall time is basically due to the lack of entries in the outstanding miss table that is used to implement a lockup-free cache. However, this scheme increases significantly the compute time for some programs like the *turb3d* (by a factor of 3 in the aggressive architecture), *mgrid* and *hydro2d*. This is due to the memory references in recurrences that limit the *II*.

5.3.3. Cache Sensitive Modulo Scheduling

In this section we propose a new algorithm, which is called *cache sensitive modulo scheduling* (CSMS), that tries to minimize both the compute time and the stall time. These terms are not independent and reducing one of them may result in an increase in the other, as we have just shown in the previous section. The proposed algorithm tries to find the best trade-off between the two terms.

The CSMS algorithm is based on early scheduling of some selectively chosen memory operations. Scheduling a memory operation using the cache-miss latency can hide almost all memory latency as we have shown in the previous section without increasing much the number of instructions (as opposed to the use of prefetch instructions). However, it can increase the execution time in three ways:

- It may increase the register pressure, and therefore, it may increase the *II* due to spill code if the performance of the loop is bounded by memory operations.
- It may increase II_{rec} because the latency of memory operations is augmented.
- It may increase the *SC* because the length of individual loop iterations may be increased. This augments the cost of the prolog and the epilog.

Two of the main issues of the CSMS algorithm is the reduction of the impact of recurrences on the *II* and the minimization of the stall time. The problem of the cost of the prolog and epilog is handled by computing two alternative schedules. Both focus on minimizing the stall time and the *II*. However, one of them reduces the impact of the prolog and the epilog at the expense of an increase in the stall time whereas the other does not care about the prolog and epilog cost. Then, depending on the number of iterations of the loop, the most effective one is chosen.

```
function CSMS(InnerLoop IL)
  return Scheduling is
  grph1 = CreateGraph(LOClatency)
  grph2 = CreateGraph(MISSlatency)
  if (RecurrencesInGraph) then
      sch1 = ComputeSchedMinRecEffect(grph1)
      sch2 = ComputeSchedMinRecEffect(grph2)
  else
      sch1 = ComputeScheduling(grph1)
      sch2 = ComputeScheduling(grph2)
  endif
  if (NITER $\leq$ UpperBound) then
      return (sch1)
  else
      return (sch2)
  endif
endfunction
```

(a) Overall algorithm

```
function ComputeSchedMinRecEffect(Graph G)
      return Scheduling is
  OrderRecurrencesByRestrictionOrder(G)
  II = $II_{res}$
  foreach (Recurrence R $\in$ G) do
      if ($II_{rec}$(R) > II) then
      II = MinimizeRecurrenceEffect(R,II)
      endif
  endforeach
  return ComputeScheduling(G)
endfunction
```

```
function MinimizeRecurrenceEffect(Rec R, int II)
  return integer is
  OrderInstructionsByLocality(R)
  while ($II_{rec}$(R) > II) do
      ChangeMostLocalityInstrLatency(R)
  endwhile
  return max (II,ComputeII(R))
endfunction
```

(b) Scheduling a loop with recurrences

Figure 5.4. CSMS algorithm

The core of the CSMS algorithm is shown in Figure 5.4 (a). The algorithm makes use of a static locality analysis (as proposed in Chapter 2) in addition to other issues in order to determine the latency to be considered when scheduling each individual instruction.

Initially, two data dependence graphs with the same nodes and edges are generated. The difference is just the latency assigned to each node. In *grph1*, each memory node is tagged according to the locality analysis: it is tagged with the cache-hit latency if it exhibits any type of locality or with the cache-miss latency otherwise. In *grph2*, all memory nodes are tagged with the cache-miss latency.

Then, a schedule that minimizes the impact of recurrences on the *II* is computed for each graph using the function *ComputeSchedMinRecEffect* that is shown in Figure 5.4 (b). The first step of this function is to order the recurrences according to the II_{rec} in decreasing order. After that, the latency of those memory operations inside recurrences that limit the *II* is changed from cache-miss to cache-hit until the *II* is limited by resources or by a more constraining recurrence (function *MinimizeRecurrence Effect*). Nodes to be modified are chosen according to a locality order, starting from the ones that exhibit most locality (the priority order is the next one: self-temporal-spatial, self-temporal, group-trailing, self-spatial, unknown and without locality).

Then, the second step is to compute the actual scheduling using the modified graph. This step can be performed through any of the software pipelined schedulers proposed in the literature.

Finally, the minimum number of iterations (*UpperBound*) that ensures that *sch2* is better than *sch1* is computed. A main difference between these two schedules is the cost of the prolog and epilog parts, which is lower for the *sch1*. This bound depends on the computed schedules and the results of the locality analysis and it is calculated through an estimation of the execution time of each schedule. The *sch1* is chosen if:

$$t_{exec_{sched1}} \leq t_{exec_{sched2}}$$

The execution time of a given schedule is estimated as:

$$t_{exec_{est}} = (NITER + SC - 1) \cdot II + t_{stall_{est}}$$

The stall time is estimated as:

$$t_{stall_{est}} = NITER \cdot \sum_{\forall op \in MEM} penalty(op) \cdot missratio(op)$$

where *penalty* is calculated as explained in Section 5.3.1:

$$penalty = LatMiss - (CycleUse - CycleProd)$$

and the *missratio* is estimated by the locality analysis. In this way, *sch1* is preferred to *sch2* if *NITER* is less or equal to:

$$\frac{(SC_2 - 1) \cdot II_2 - (SC_1 - 1) \cdot II_1}{(II_1 - II_2) + \sum_{\forall op \in MEM} (penalty_1(op) - penalty_2(op)) \cdot missratio(op)}$$

We use a scheduling according to the locality a not the CHL (which achieves the minimum SC) in order to take into account the possible poor locality of some loops.

5.4. Evaluation

In this section we present a performance evaluation of the CSMS algorithm. We compare its performance to that of the basic schemes evaluated in Section 5.3.2. It is also compared with some alternative binding (early scheduling) and nonbinding (inserting prefetch instructions) prefetch schemes.

5.4.1. Architecture model

A VLIW machine has been considered to evaluate the performance of the different scheduling algorithms. We have modeled two architectures in order to evaluate different aspects of the produced schedulings such as execution time, stall time, spill code, etc. The first architecture is called *simple* and it is composed of four functional units: integer, floating point, branch and memory. The cache-miss latency for the first level cache is 10 cycles. The second architecture is called *aggressive* and it has two functional units of each type and the cache-miss latency is 20 cycles. All functional units are fully pipelined except divide and square root operations. In both models the first memory level corresponds to a 8KB lockup-free, direct-mapped cache with lines of 32 bytes and 8 outstanding misses. Other features of the modeled architectures are depicted in Table 5.1.

MACHINE MODEL	Simple	Aggressive
Integer FUs	1	2
FP FUs	1	2
Branch FUs	1	2
Memory FUs	1	2
Cache Size	8 Kb	
Line Size	32 bytes	
Outstanding misses	8	
Memory latency	1/10	1/20
Number of registers	32	

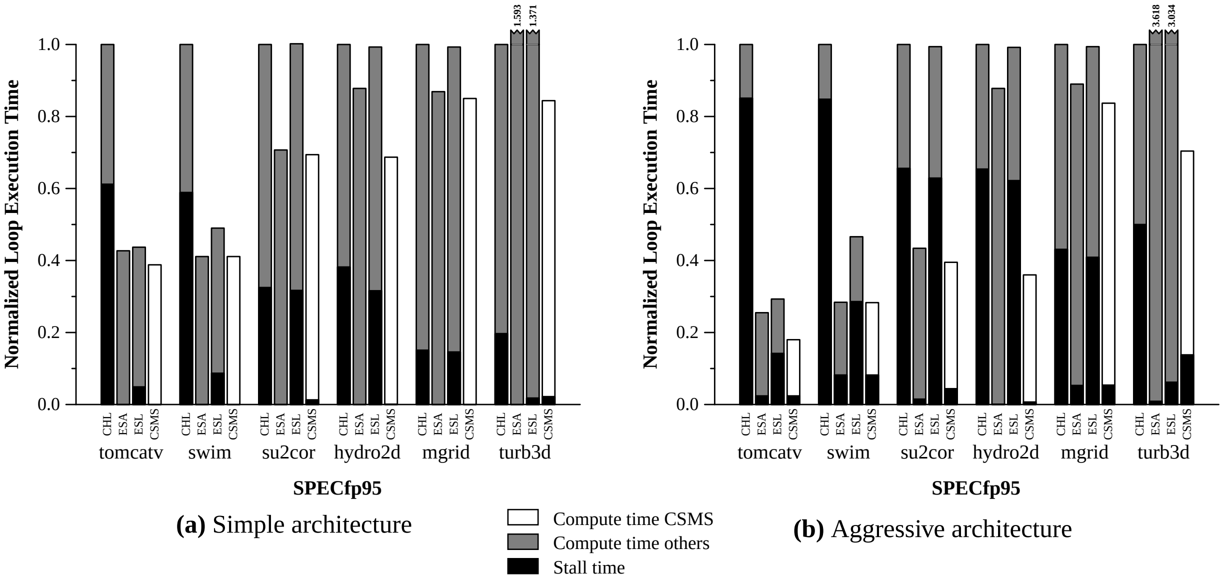
	Other instructions	Latency
Integer	**ARITH**	2
	MUL	4
	DIV or POW	6
Floating Point	**ARITH**	4
	MUL	8
	DIV or SQRT or POW	12
	TRIG	2
Control	**JUMP**	1
	BRANCH	2
	CALL or RETURN	4

Table 5.1. Modeled architectures

In the modeled architectures there are two reasons for the processor to stall: (a) when an instruction requires an operand that is not available yet (e.g., it is being read from the second level cache), and (b) when a memory instruction produces a cache miss and there are already 8 outstanding misses.

5.4.2. Experimental framework

The locality analysis and scheduling task have been performed using the ICTINEO toolset [4]. After translating the code to such low-level representation and applying classical optimizations, the dependence graph of each innermost loop is constructed according the particular prefetching approach. Then, instructions are scheduled using any software pipelining algorithm. The particular software pipelining algorithm used in the experiments reported here is the HRMS [65], which has been shown to be very effective to minimize both the *II* and the register pressure.



(a) Simple architecture

(b) Aggressive architecture

Figure 5.5. CSMS algorithm compared with early scheduling

The resulting code is instrumented to generate a trace that feeds a simulator of the architecture. Each program was run for the first 100 million of memory references. The performance figures shown in this section refer to the innermost loops (without subroutine calls) contained in this part of the program. We have measured that memory references inside such loops represent about 95% of all the memory instructions considered for each benchmark, so the statistics are quite representative of the whole section of the program.

The different prefetching algorithms have been evaluated for the following SPECfp95 benchmarks: *tomcatv*, *swim*, *su2cor*, *hydro2d*, *mgrid* and *turb3d*.

5.4.3. Early scheduling

In this section we compare the CSMS algorithm with other schemes based on early scheduling of memory operations. These schemes are: (i) use always cache-hit latency (CHL), (ii) use always cache-miss latency (ESA), and (iii) schedule instructions that have some type of locality using the cache-hit latency and schedule the remaining ones using the cache-miss latency. This later scheme will be called *early scheduling according to locality* (ESL).

The different algorithms have been evaluated in terms of execution time, which is split into compute and stall time. The stall time is due to dependences or to the lack of entries in the outstanding miss table. In Figure 5.5 we can see the results for both the simple and the aggressive architectures. For each benchmark all columns are normalized to the CHL execution time. It can be seen that the CSMS algorithm achieves a compute time very close to the CHL scheme whereas it has a stall time very close to the ESA scheme. That is, it results in the best trade-off between compute and stall time. In programs where recurrences limit the initiation interval, and therefore the ESA scheme increases the compute time (for instance in *hydro2d* and *turb3d* benchmarks), the CSMS method minimize this effect at the expense of a slight increase in the stall time.

The CSMS scheme increases the register pressure when compared with the CHL method. This results in an increase of 0.1% and 20% of the spill code for the simple and aggressive architectures respectively. However, the penalty of this additional spill code in much lower than the reduction in the stall time.

SPECfp95	SIMPLE ARCHITECTURE			AGGRESSIVE ARCHITECTURE		
	ESA	ESL	CSMS	ESA	ESL	CSMS
tomcatv	2.34	2.28	2.57	3.92	3.41	5.56
swim	2.43	2.04	2.43	3.52	2.14	3.52
su2cor	1.41	0.99	1.44	2.30	1.00	2.53
hydro2d	1.13	1.00	1.45	1.13	1.00	2.78
mgrid	1.15	1.00	1.17	1.12	1.00	1.19
turb3d	0.62	0.73	1.18	0.27	0.33	1.42
HARMONIC MEAN	**1.22**	**1.13**	**1.54**	**0.93**	**0.88**	**2.17**

Table 5.2. Relative speed-up

Table 5.2 shows the relative speed-up of the different schedulers with respect the CHL scheme. On average, all alternative schedulers outperform the CHL scheme (which is usually the one used by software pipelining schedulers). However, for some programs (mainly for *turb3d)* the ESA and ESL schedulers perform worse than the CHL due to the increase in the *II* caused by recurrences. The CSMS algorithm achieves the best performance for all benchmarks. For the simple architecture the average speed-up is 1.61, and for the aggressive architecture it is 2.47.

Table 5.3 compares the CSMS algorithm with an optimistic lower execution time (LBND) as defined in Section 5.3.2 that is used as a lower bound of the execution time. It also shows the percentage of the execution time that the processor is stalled. It can be seen that for the simple architecture the CSMS algorithm is close to the optimistic bound and it does not cause almost any stall. For the aggressive architecture, the performance of the CSMS is worse than that of LBND and the stall time represents about 10% of the total

SPECfp95	SIMPLE ARCHITECTURE		AGGRESSIVE ARCHITECTURE	
	LBND/ CSMS	%Stall	LBND/ CSMS	%Stall
tomcatv	0.998	0.02	0.830	13.23
swim	1.000	0.00	0.537	28.87
su2cor	0.972	1.92	0.873	11.17
hydro2d	0.978	0.18	0.962	1.84
mgrid	0.998	0.05	0.680	6.39
turb3d	0.951	2.54	0.709	19.54
HARMONIC MEAN	**0.982**	**0.00**	**0.737**	**6.31**

Table 5.3. CSMS compared with LBND scheduling

execution time. Note however, that the lower bound could be quite below the actual minimum execution time. Table 5.4 compares the different schemes using the CHL algorithm as a reference point.

SPECfp95	SIMPLE ARCHITECTURE						AGGRESSIVE ARCHITECTURE					
	ESA		ESL		CSMS		ESA		ESL		CSMS	
	∆Compute	∇Stall	∆Compute	∇Stall	∆Compute	∇Stall	∆Compute	∇Stall	∆Compute	∇Stall	∆Compute	∇Stall
tomcatv	10.05	100.00	0.00	91.95	0.00	99.98	55.03	97.21	1.34	83.36	4.69	97.20
swim	0.00	100.00	-1.94	85.19	0.00	100.00	32.89	90.27	18.42	66.25	32.23	90.34
su2cor	4.74	100.00	1.48	2.54	0.88	95.90	21.80	97.67	6.10	4.09	2.03	93.27
hydro2d	42.07	99.99	9.54	3.79	11.00	99.62	153.46	99.85	6.93	4.84	2.02	98.98
mgrid	2.35	99.89	-0.23	3.59	0.11	99.68	47.10	87.58	2.81	5.19	37.60	87.57
turb3d	98.38	94.35	68.49	85.75	2.36	94.35	621.80	98.21	494.40	87.59	13.20	72.48
GEOMETRIC MEAN	**2.78**	**99.01**	**0.87**	**16.88**	**0.12**	**98.22**	**74.95**	**95.02**	**10.69**	**19.17**	**8.22**	**89.51**

Table 5.4. Increment of compute time and decrement of stall time in relation to the CHL (in percentage)

For each scheme it shows the increase in compute time and the decrease in stall time. As we have seen before, scheduling memory operations using the cache-miss latency can affect the initiation interval and the stage count, which results in an increase in the compute time. The column denoted as $\Delta Compute$ represents the increment in compute time compared with the CHL scheduling. For any scheme s, it is calculated as:

$$\Delta Compute = \frac{\langle t_{exec_s} - t_{stall_s} \rangle - (t_{exec_{CHL}} - t_{stall_{CHL}})}{t_{exec_{CHL}} - t_{stall_{CHL}}} \times 100$$

The stall time due to dependences can be eliminated by scheduling memory instructions using the cache-miss latency. By default, spill code is scheduled using the cache-hit latency and therefore it may cause some stalls, although it is unlikely because the spill code usually is a store followed by a load to the same address. Since usually they are not close (otherwise the spill code hardly reduces the register pressure), the load will cause a stall only if it interferes with a memory reference in between the store and itself. The column denoted as $\nabla Stall$ represents the percentage of the stall time caused by the CHL algorithm that is avoided. For any scheme *s*, it is calculated as:

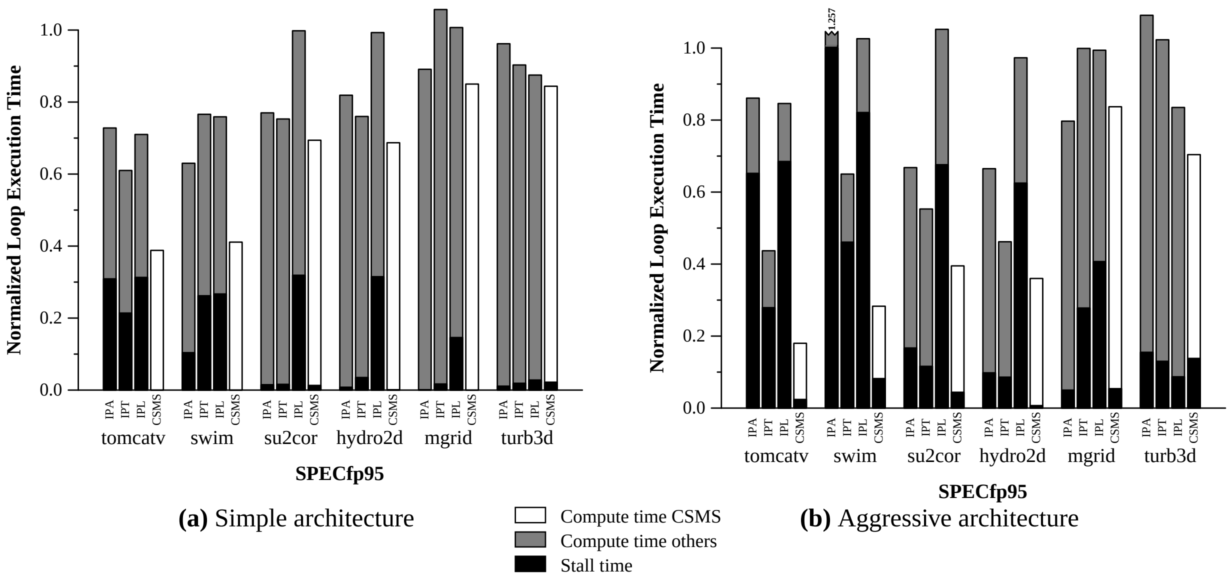
$$\nabla Stall\ (\%) = \frac{t_{stall_{CHL}} - t_{stall_s}}{t_{stall_{CHL}}} \times 100$$

We can see in Table 5.4 that the CSMS algorithm achieves the best trade-off between compute time and stall time, which is the reason for outperforming the others. The ESA scheme is the best one to reduce the stall time but at the expense of a large increment in compute time, mainly when the architecture becomes more aggressive.

5.4.4. Inserting prefetch instructions

In order to reduce the penalties caused by memory operations, an alternative to early scheduling of memory instructions is inserting prefetch instructions, which are provided by many current instruction set architectures (e.g., the *touch* instruction of the PowerPC [19]). This new scheme can introduce additional spill code since it increases the register pressure. In particular, the lifetime of values that are used to compute the effective address is increased since they are used by both the prefetch and ordinary memory instructions. It can also increase the initiation interval due to additional memory instructions.

We have evaluated three alternative schemes to introduce prefetch instructions: (i) insert prefetch always (IPA), (ii) insert prefetch for those references without temporal locality even if they exhibit spatial locality, according to the static locality analysis (IPT), and (iii) insert prefetch for those instructions without any type of locality (IPL). The first scheme is expected to result in a very few stalls but it requires many additional instructions, which may increase the *II*. The IPT scheme is more selective when adding prefetch instruction. However, it adds unnecessary prefetch instructions for some references with just spatial locality. Instructions with only spatial locality will cause a cache miss only when a new cache line is accessed if it is not in cache. The IPL scheme is the most conservative in the sense that it adds the less number of prefetch instructions.



(a) Simple architecture

(b) Aggressive architecture

Figure 5.6. CSMS algorithm compared with inserting prefetch instructions

In Figure 5.6 it is compared the total execution time of the CSMS scheduling against the above-mentioned prefetching schemes. The figures are normalized to the CHL scheduling. The CSMS scheme always performs better than the schemes based on inserting prefetch instructions except for the *mgrid* benchmark in the aggressive architecture. In this latter case, the IPA scheme is the best one but the performance of the CSMS is very close to it.

Among the schemes that insert prefetch instructions, none of them outperforms the others in general. Depending on the particular program and architecture, the best one among them is a different one. The prefetch schemes outperform the CHL scheme in general (i.e., the performance figures in Figure 5.6 are in general lower than 1) but in some cases they may be even worse than the CHL, which is in general worse than the schemes that are based on early scheduling.

Comparing binding (Figure 5.5) with nonbinding (Figure 5.6) schemes, it can be observed that binding prefetch is always better for the three first benchmarks. Both schemes have similar performance for the next two benchmarks and only for the last one, nonbinding prefetch outperforms the binding schemes.

To understand the reasons for the behavior of the prefetch schemes, we present below some additional statistics for the aggressive architecture. Table 5.5 shows the percentage of additional memory instructions that are executed for the CSMS algorithm and for those schemes based on inserting prefetch instructions. In the CSMS algorithm, additional instructions are only due to spill code whereas in the other schemes they are due to spill code and prefetch instructions. We can see in this table that, except for the IPL scheme for the *mgrid* benchmark, the prefetch schemes require much higher number of additional memory instructions. As expected, the increase in number of memory instructions of the IPA scheme is the highest, followed by IPT, then the IPL and finally the CSMS.

SPECfp95	AGGRESSIVE ARCHITECTURE			
	CSMS	INSERTING PREFETCH INSTR.		
		IPA	IPT	IPL
tomcatv	32.12	60.99	50.29	53.84
swim	38.75	64.00	48.47	44.77
su2cor	0.00[a]	60.52	48.71	23.34
hydro2d	2.12	55.49	39.94	2.85
mgrid	49.90	59.26	56.57	7.50
turb3d	0.00[a]	69.16	49.19	51.34
ARITHMETIC MEAN	**20.48**	**61.57**	**48.86**	**30.60**

Table 5.5. Percentage of additional memory references

[a] There is spill code, but not in the simulated part of the program.

Table 5.6 shows the increase in compute time and the decrease in stall time of the schemes based on inserting prefetch instructions in relation to the CHL scheme. Negative numbers indicate that the stall time is increased instead of decreased.

SPECfp95	SIMPLE ARCHITECTURE						AGGRESSIVE ARCHITECTURE					
	IPA		IPT		IPL		IPA		IPT		IPL	
	ΔCompute	∇Stall	ΔCompute	∇Stall	ΔCompute	∇Stall	ΔCompute	∇Stall	ΔCompute	∇Stall	ΔCompute	∇Stall
tomcatv	7.98	49.50	2.06	65.03	2.31	48.85	40.26	23.38	6.04	67.21	8.05	19.50
swim	27.98	82.34	22.62	55.51	19.70	54.66	65.13	-18.63	24.34	45.63	34.86	3.18
su2cor	11.85	95.38	9.18	95.07	0.59	1.84	45.63	74.54	27.03	82.31	9.30	-3.04
hydro2d	31.22	97.90	17.31	90.83	9.70	17.53	63.87	85.01	8.67	86.85	0.57	4.43
mgrid	4.82	99.33	22.49	88.74	1.41	3.31	31.28	88.39	26.71	35.49	3.16	5.56
turb3d	18.43	94.41	10.08	90.35	5.47	85.78	87.20	69.00	78.60	74.00	49.60	82.60
GEOMETRIC MEAN	**13.94**	**84.22**	**10.90**	**79.37**	**3.552**	**17.04**	**52.46**	**NaN**	**20.41**	**62.14**	**7.84**	**NaN**

Table 5.6. Increment of compute time and decrement of stall time for schemes based on inserting prefetch instructions (in percentage)

We can see in Table 5.6 that the compute time is increased by prefetching schemes since the large number of additional instructions may imply a significant increase in the *II* for those loops that are memory bound. The stall time is in general reduced, but the reduction is less than that of the CSMS scheme (see Table 5.4). The program *mgrid* is the only one for which there is a prefetch based scheme (IPA) that outperforms the CSMS algorithm. However, the difference is very slight and for the remaining programs the performance of the CSMS scheme is overwhelmingly better than that the IPA scheme. Table 5.7 shows the miss ratio of the different prefetching schemes compared with the miss ratio of a nonprefetching scheme (CHL).

SPECfp95	AGGRESIVE ARCHITECTURE			
	CHL	IPA	IPT	IPL
tomcatv	68.02	41.08	49.71	43.37
swim	64.65	31.55	44.18	51.56
su2cor	25.43	2.35	5.68	21.55
hydro2d	19.57	1.33	5.04	18.80
mgrid	6.46	0.57	2.91	5.35
turb3d	10.68	2.11	2.39	2.64
GEOMETRIC MEAN	**23.07**	**4.11**	**8.71**	**15.29**

Table 5.7. Miss ratio for the CHL and the different prefetching schemes

We can see that in general the schemes that insert most memory prefetches produce the highest reductions in miss ratio. However, inserting prefetch instructions do not remove all cache misses, even for the scheme that inserts a prefetch for every memory instruction (IPA). This is due to cache interferences between prefetch instructions before the prefetched data is used. This is quite common in the programs *tomcatv* and *swim*. For instance, if two memory references that interfere in the cache are very close in the code, it is likely that the two prefetches corresponding to them are scheduled before both memory references. In this case, at least one of the two memory references will miss in spite of the prefetch. Besides, if the prefetches and memory instructions are scheduled in reverse order (i.e., instruction A is scheduled before B but the prefetch of B is scheduled before the prefetch of A), both memory instructions will miss.

To summarize, there are two main reasons for the bad performance of the schemes based on inserting prefetch instructions when compared with the CSMS scheme:

- They increase the compute time due to the additional prefetch instructions and spill code.
- They are not always effective in removing stalls caused by cache misses due to interferences between the prefetch instructions.

5.5. CHAPTER SUMMARY

The interaction between software prefetching and software pipelining techniques for VLIW architectures has been studied. We have shown that modulo scheduling schemes using cache-hit latency produce many stalls due to dependences with memory instructions. For a simple architecture the stall time represents about 32% of the execution time and 63% for an aggressive architecture. Thus, ignoring memory effects when evaluating a software pipelined scheduler may be rather inaccurate.

We have compared the performance of different prefetching approaches based on either early scheduling of memory instructions (binding prefetch) or inserting prefetch instructions (nonbinding prefetch). We have seen that both provide a significant improvement in general. However, methods based on early scheduling outperform those based on inserting prefetches. The main reasons for the worse performance of the latter methods are the increase in memory pressure due to prefetch instructions and additional spill code, and their limitation to remove short-distance conflict misses.

We have proposed an heuristic scheduling algorithm (CSMS), which is based on early scheduling of some memory instructions, that tries to minimize both the compute and the stall time. The algorithm makes use of a static locality analysis to schedule instructions in recurrences. We have shown that it outperforms the rest of strategies. For instance, when compared with the approach based on scheduling memory instructions using the cache-hit latency, the produced code is 1.6 times faster for a simple architecture, and 2.5 times faster for an aggressive architecture. In the former case, we have also shown that the execution time is very close to an optimistic lower bound.

6

INSTRUCTION SCHEDULING FOR CLUSTERED VLIW ARCHITECTURES

Clustered organizations are becoming a common trend in the design of VLIW architectures. In this chapter we first propose a novel modulo scheduling approach for architectures in which both register file and functional units are partitioned. The proposed technique performs the cluster assignment and the instruction scheduling in a single pass, which is shown to be more effective than doing first the assignment and later the scheduling. We also show that loop unrolling significantly enhances the performance of the proposed scheduler, especially when the communication channel among clusters is the main performance bottleneck. By selectively unrolling some loops, we can obtain the best performance with the minimum increase in code size. Performance evaluation shows that the clustered architecture achieves about the same IPC (Instructions Per Cycle) as an equivalent unified architecture with the same resources. Then, the algorithm is extended to clustered architectures with a distributed data cache. The proposed algorithm takes into account both register and memory inter-cluster communications. It has been evaluated for both 2- and 4-cluster configurations and for different number and latency of inter-cluster buses. It is shown that the proposed algorithm produces schedules with very low communication requeriments and it outperforms previous cluster-oriented schedulers.

6.1. INTRODUCTION

Semiconductor technology has experienced a continuous improvement in the past and current projections anticipate that this trend will continue in the forthcoming years [93]. By reducing the minimum feature size, new technologies will pack more logic in a single chip but new problems may arise. Technology projections point out that wire delays will be one of the main hurdles for improving instruction throughput of future microprocessors [93]. As wire delays grow relative to gate delays and feature sizes shrink, the percentage of on-chip transistors that can be reached in a single cycle will decrease, and microprocessors will become *communication bound* rather than *capacity bound* [2][71].

In the same way, an approach to enhance the processor performance is to exploit more instruction-level parallelism (ILP). However, this requires more functional units, registers and more resources in general. This increment in resources can affect the cycle time of the processor. For instance, Palacharla et al. [80] showed that the bypass delay and the register file access time are some of the critical delays of current microprocessors.

Proposed approaches to deal with these problems are based on exploiting communication locality. The basic idea is to divide the system into several processing "units" that can work almost independently and at a very high frequency. Then, some communication channels are included in order to exchange signals/data among "units". This partition of the processor in quasi-independent units is nowadays called *clustering*.

Current trends in clustering focus on the partition of the register file. Functional units are grouped and assigned to a register file partition so they can only read their operands from their local register file. Values generated by one cluster and needed by another must be communicated. In this way, both bypasses among functional units and ports of the register file are reduced as well as the number of registers of each local register file. Clustered designs can be found in current research proposals (multiscalar [29][95], multithreading [69], trace processors [88][106], etc.) and even in some commercial superscalar processors such as the Alpha 21264 [39]. However, this trend is even more common for VLIW processors used in the embedded/DSP domain. Examples of the latter are the Texas Instrument's TMS320C6000 [101], the Equator's MAP1000 [68], the Analog's TigerSharc [30] and the HP/ST's Lx plattform [25].

In this chapter we focus on clustered VLIW architectures. As previously mentioned in Chapter 5, software pipelining is a very effective technique to statically schedule loops. The most popular scheme to perform software pipelining is called modulo scheduling. In this chapter we first propose a cluster-oriented modulo scheduling algorithm for an architecture which has all the resources partitioned. For the sake of simplicity, we first consider a clustered architecture with a shared cache memory and propose an algorithm for reducing inter-cluster register communication and maximizing workload balance. By performing the cluster assignment and the instruction scheduling at the same time and by using loop unrolling, the proposed technique can hide practically all the communication latency, resulting in an IPC very similar to that of a unified architecture with the same resources, for different communication delays and bandwidths. When the cycle time is factored in, the cluster architecture achieves an average speed-up of 3.6 for the SPECfp95 on a 4-cluster configuration.

Then, we consider a clustered VLIW microarchitecture with a distributed cache memory. This architecture has all the resources distributed: instruction fetch, execute and memory units. It resembles very much a multiprocessor, with the exception that all the clusters progress in a lockstep mode, and inter-cluster register communications are controlled by the compiler by means of certain fields in the ISA. Because of this resemblance we refer to this architecture as a *multiVLIWprocessor*.

The effectiveness of this microarchitecture strongly depends on the ability of the compiler to generate code that balances the workload of the different clusters and result in few inter-cluster communications. We propose a modulo scheduler for *multiVLIWprocessors* that includes some heuristics for minimizing inter-cluster register communication, based on the information provided by the data dependence graph. Besides, it implements a powerful memory locality analysis based on *Cache Miss Equations* [33] as described in Chapter 2, which guides the scheduling of memory instructions with the objective of minimizing/hiding inter-cluster memory communications.

6.2. PREVIOUS WORK

There are several works related with instruction scheduling for clustered architectures. The first proposal for solving the problem of scheduling instructions for partitioned register files is in the work by Ellis in a prototype compiler called Bulldog [24]. That work implements trace scheduling and decides cluster assignments to the instructions in the trace. In that algorithm cluster selection and list scheduling are treated as two sequential phases. The cluster assignment step uses a BUG algorithm (Bottom-Up Greedy). Communication operations are inserted during the scheduling step if necessary.

Capitanio et al. present a scheduling algorithm [11] whose objective is code partition when the VLIW clustered architecture does not have full connectivity among all registers and functional units. The algorithm strategy also performs cluster assignment and instruction scheduling in two sequential phases.

Jang et al. [50] present another scheduling scheme that uses separate assigning/scheduling phases. In their work, a graph is partitioned using a k-way partitioning algorithm (where k is the number of clusters). Their main aim is to achive a balanced scheduling. In the dependence graph each node represents a register (or value) instead of an operation in order to provide flexibility in their retargetable compiler.

These works differ from the approach presented in this chapter in two basic aspects: they focus on scheduling instructions in acyclic codes (more particularly, they do not deal with modulo scheduling) and follow an approach where cluster assignment and instruction scheduling are performed in two sequential steps.

Özer et al. [78] propose a scheduling algorithm called unified-assign-and-scheduling (UAS) that differs from previous approaches to scheduling instructions. Instead of first partitioning the instructions among the clusters and then scheduling them, these two steps are performed at the same time. The algorithm proposed in this paper follows the same strategy. However, our work focuses on modulo scheduling instead of list scheduling.

There are a couple of works related to modulo scheduling for clustered architectures. Nystrom and Eichenberger [77] present an algorithm to assign nodes to clusters when modulo scheduling is performed. Their algorithm deals with cases where the connection among the different register files is bus-based or grid-based. Their approach follows a strategy where the cluster assignment and node scheduling correspond to different phases. If any of them fails, the algorithm is re-started by incrementing the initiation interval. They focus on two main aspects: the impact of loop-carried dependences and the negative impact of aggressively filling clusters. They obtain good results for the loops evaluated but their architecture almost never saturates the communication channels (because they assume sufficient low-latency buses), and thereby the effect of communication is very low. However, as we will see in a later section, when the number of channels (buses in our case) decreases or the communication latency increases, the performance of this algorithm is significantly degraded.

Fernandes et al. [28] propose an approach to perform both scheduling and partitioning in a single step for software pipelined loops. However, they assume an architecture with an unusual register file organization based on a set of local queues for each cluster and a queue file for each communication channel.

There are also some works that schedule instructions dynamically among the different clusters of functional units for a variety of architectures. Some interesting works are [56][27][92][69][10].

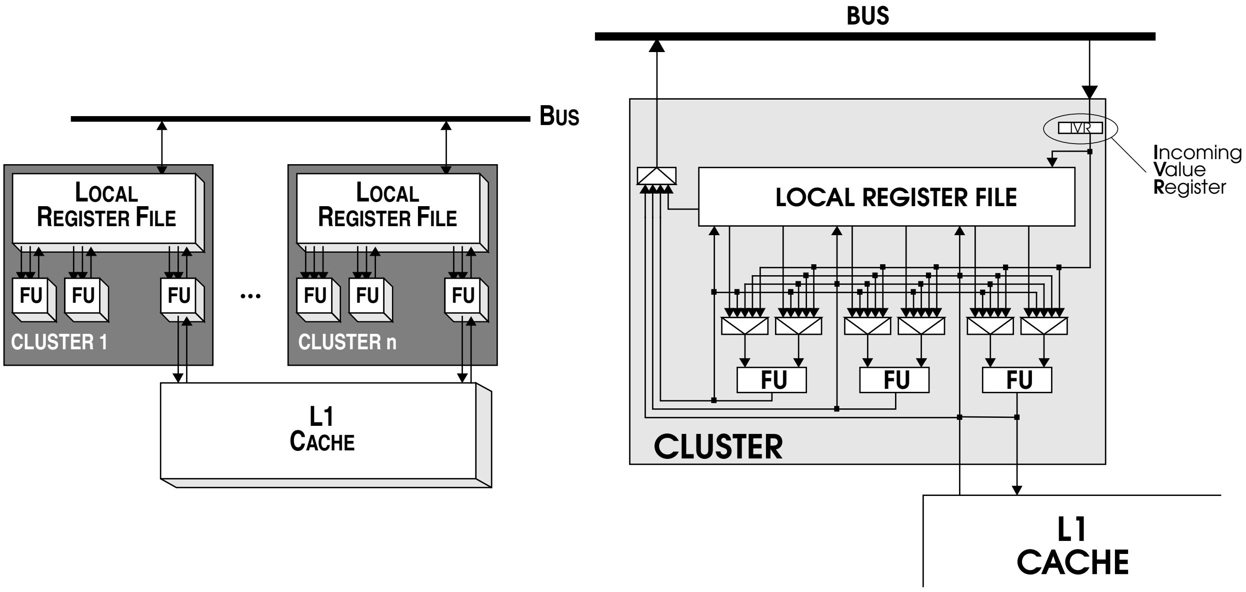


Figure 6.1. VLIW clustered architecture and detailed architecture of a single cluster

6.3. Scheduling for a Semi-Distributed Architecture

In this section we first describe the clustered architecture, including the VLIW instruction format which incorporates explicit control of inter-cluster register communications. We then present the scheduling algorithm that includes selective loop unrolling.

6.3.1. Architecture

The clustered VLIW architecture that we assume in this section is shown in Figure 6.1. It is composed of different clusters, each one made up of different functional units and a local register file. Values generated by one cluster and consumed by another are communicated through a bus shared by all the clusters. The architecture may have one or several buses in order to communicate values among the different clusters. When a value is communicated, the employed bus is busy during the latency of the communication. The cluster that writes onto the bus and the cluster/s that read from the bus are codified in the VLIW instruction, as described below. All the clusters also share the memory hierarchy, starting from the L1 cache. In this work we have considered that all clusters are homogeneous (i.e., same number of registers and type/ number of functional units) although the proposed scheduling techniques can easily be generalized for non-homogeneous configurations.

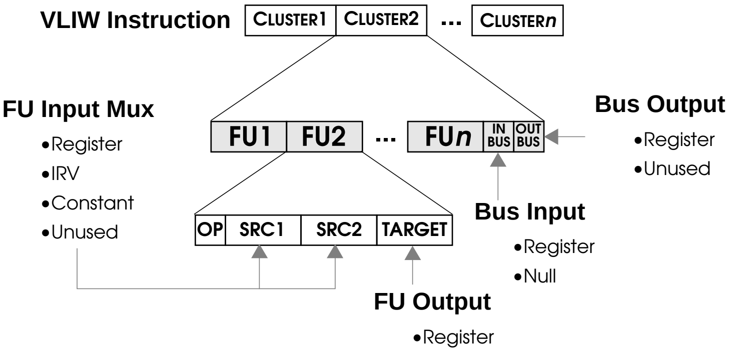


Figure 6.2. VLIW instruction format

The detailed architecture of a single cluster is also shown in Figure 6.1. The inputs of each functional unit are multiplexed among a value read from the local register file, values obtained through bypasses from other functional units of the same cluster, and finally the value that comes from a bus. This last value is stored in a special register called incoming value register (IRV), and can feed a functional unit and/or be stored in the local register file (in the case that another instruction scheduled in this cluster needs the value later). On the other hand, the data that is placed on the bus can be either obtained from the output of a functional unit or from the local register file.

Register values generated by one cluster and needed by another one are communicated through a set of buses that are shared by all clusters (called *register buses*). A value that is put in a register bus can come from either the local register file or the output of a functional unit through a short-circuit. On the other hand, a value that is read from the bus can be stored in a register file, feed a functional unit or both. Thus, instruction register operands can be read from either the local register file or any bus, and instruction results can be written into the register file and to any register bus. All register communication operations are explicitly encoded in the appropriate fields of the VLIW instruction, which are set at compile time. Thus, no additional hardware is needed to manage and arbitrate register buses. The detailed VLIW instruction format is shown in Figure 6.2. A stall in one cluster affects all the others, so that all the clusters work on the same VLIW instruction. Each instruction for a particular cluster consists of the following fields. An operation for each functional unit in that particular cluster (FU_j) and the source (IN BUS) and target (OUT BUS) of the bus (there are as many IN/OUT fields as number of buses). The IN BUS field indicates, if necessary, the register in the local register file in which the value in IRV has to be stored. IRV (*Incoming Register Value*) is a special register in each cluster that latches the value that comes from the bus. The OUT BUS field indicates from which local register a value has to be issued to the bus, if any. If the register is being written in that cycle, the data will be bypassed from the output of the corresponding functional unit. As a bus is a resource shared by all the clusters, when one particular cluster places a data on the bus (OUT BUS), this bus will be busy during the entire bus latency and no other instruction can use this bus meanwhile (a bus is considered by the scheduling algorithm as another resource in the reservation table).

6.3.2. Basic Scheduling Algorithm

In this section we present the proposed modulo scheduling algorithm for semi-distributed clustered VLIW architectures. We first present a basic scheduling algorithm, which tries to reduce the penalties of inter-cluster communications as its main goal, since the buses are the most constrained resource for many loops. However, this kind of algorithms are not sufficient for many loops (since many communications cannot be hidden). Therefore, in next section we also present an algorithm for unrolling some loops in order to further reduce the impact of communications on the final scheduling.

Algorithm

The main objective of the *Basic Scheduling Algorithm* (BSA) is to reduce the number of communications or, in other words, obtain the same II as an hypothetical unified architecture (that is, without clustering and with the same number of resources). Our algorithm employs a unified assign-and-schedule approach, as proposed by Özer et al. [78] for non-cyclic scheduling, where the cluster selection heuristics prioritize those clusters that minimize the number of communications.

The scheduling algorithm is shown in Figure 6.3. In the first step of the algorithm (1) a list with all the nodes of the graph is built (which represent instructions). In this list, all nodes are sorted in order to reflect the sequence to follow during the scheduling phase. We have chosen the ordering performed by the SMS [66]. This ordering gives priority to the nodes in recurrences with the highest *RecMII* (that is, according to their criticallity). *RecMII* stands for the minimum initiation interval constrained by recurrences. Besides, the resulting order ensures that a node in a particular position of the list only has predecessors or successors before it (except in the case of sorting a new subgraph). Moreover, nodes that are neighbors in the graph are placed close together in the ordering.

Once the nodes have been sorted, and following this ordering, each one is scheduled in the appropriate cycle and cluster. If the current node does not have a predecessor nor a successor, the default cluster (`def-cluster` variable) is set to the next one according to a circular order (2). Other possibilities for selecting the default cluster are feasible, such as choosing the least loaded one.

```
(1)NLIST = OrderNodes(G);
   foreach (n in NLIST) do {
     // Check if it is a new subgraph
(2)  if (!SchedPred(n, G) && !SchedSucc(n, G))
       defcluster = NextCluster(defcluster);
     // Compute the profit contributed in outedges
(3)  foreach (c in CLIST) do {
       tmpoutedges = TryNodeOnCluster(n, c, G);
       profit[c] = OutEdgesOnCluster(c) - tmpoutedges;
     }
     // Build a list with the best ones
(4)  candlist = ChooseBestProfit(profit);
     // Choose the most appropriate
(5)  if (ListLenght(candlist) == 0) {
       II++;
       ReInitialize();
     }
     if (ListLenght(candlist == 1)
(6)    chosen = ChooseCluster(candlist);
     else {
(7)    if (n = ExistPredOrSuccInCand(candlist))
         chosen = n;
       else {
(8)      if (candlist[defcluster] == Ok)
           chosen = defcluster;
         else
(9)        chosen = MinimizeRegRequirements(candlist);
       }
     }
(10)ScheduleNode(n, chosen);
   }
```

Figure 6.3. Basic scheduling algorithm

The core of the algorithm is in fragment (3). In this loop we attempt to schedule the current node in each possible cluster (i.e. those clusters with an empty slot for the corresponding functional unit). Those clusters for which the insertion of this node would increase the register requirements above the number of available registers are discarded[1]. The variable `tmpoutedges` represents the number of edges from the nodes scheduled in the candidate cluster (including the current node) to the rest of nodes. This measure represents the number of communications needed in this cluster if the schedule would finish here. The idea of our algorithm is to schedule a node in the cluster that results in the best use of outedges. For this reason the profit in a cluster (`profit[c]`) is defined as the difference between the outgoing edges before and after scheduling the current node in this cluster. Then, a list of the clusters with the highest profit is built (4). If no cluster is in the list (all the slots of the functional units are full, or none of the registers nor buses are available), then the initiation interval is increased and the whole process is reinitialized (5). Otherwise, one

[1] Insertion of spill code may improve the performance of the proposed algorithm. Techniques to insert spill code on the fly can be found elsewhere [54] and are beyond the scope of this work.

cluster is chosen according to the next prioritized criteria: the only one (6), the cluster with any predecessor or successor (if any) of the current node (7), the `defcluster` (8), or the one that minimizes the register requirements (9). Once the cluster is chosen, the node is scheduled in the appropriate cycle and both functional unit and bus (if needed) are marked as occupied in the reservation table (10).

Note in particular the following cases:

a) The first node of a new subgraph is being scheduled: as it has no successor nor predecessor already scheduled, the benefit in outedges is the same for all the clusters. Therefore, the chosen cluster is the default one.

b) If the loop has been unrolled and a node of a particular iteration is being scheduled and the node does not have any dependence with nodes in other iterations, the benefit will be maximized if it is scheduled in the same cluster as the other nodes of the same iteration.

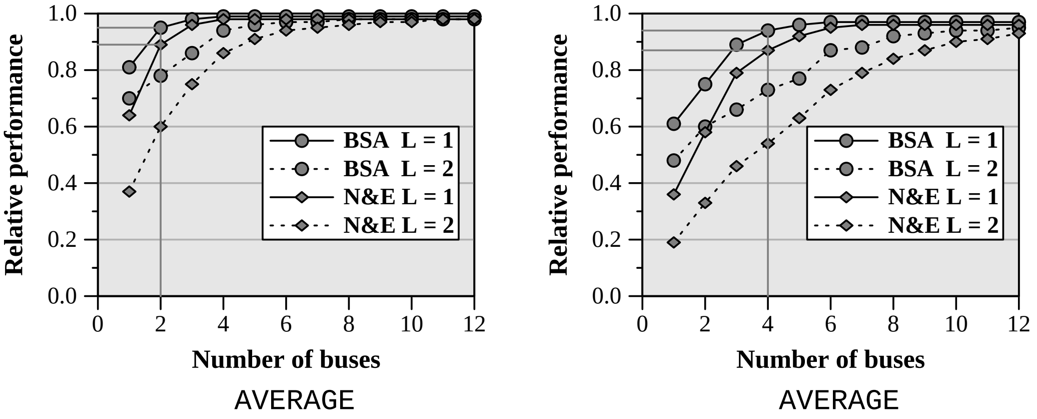
Therefore, this algorithm tries to schedule subgraphs that are disconnected in different clusters, and in particular, iterations of an unrolled loop follow this trend.

Evaluation

For a clustered VLIW architecture, both *II* and *SC* can be affected by inter-cluster communications. If the communication buses become saturated, a higher *II* is required. On the other hand, communication operations may increase the length of the schedule, and therefore the *SC* may be increased. Thus, the IPC of a VLIW clustered architecture will be lower than that of a VLIW unified architecture with the same resources in general.

In this section we show how the number and latency of buses affect the final modulo scheduling in a VLIW clustered architecture compared to a unified architecture with the same resources (functional units and registers). We also highlight the differences between approaches that perform first the partitioning of instructions among clusters and then compute the schedule for each cluster and approaches that do both tasks simultaneously. In general, the latter type of methods will be better, since the partitioning may benefit from information obtained from the partial schedule.

Figure 6.4 shows performance results relative to a unified machine obtained through simulation compared with an hypothetical unified machined. It shows the performance of the basic algorithm we propose based on a unified assign-and-schedule strategy and the algorithm proposed by Nystrom and Eichenberger [77], which consists of a first phase for performing the graph partitioning and a second phase for schedul-



(a) 2-cluster configuration **(b)** 4-cluster configuration

Figure 6.4. Relative performance of VLIW clustered architectures assuming the same cycle time

ing each node in the corresponding cluster. For the latter approach we have used the cluster assignment algorithm that they proposed and then, we have used the SMS instruction scheduler [66].

Graphs on the left show the results for a 2-cluster configuration whereas on the right the results are for a 4-cluster configuration (see Section 6.3.4 for more details about each particular architecture and the bechmarks evaluated). In these figures, we can see the relative performance averaged for all evaluated benchmarks. In these two figures we can also see the results of our basic scheduling algorithm (BSA, lines marked with circles) and Nystrom et al. (N&E, lines marked with diamonds) assuming buses with a latency of one (L=1, solid line) and two (L=2, dotted line) cycles.

We can see in these figures that assuming the same configurations (clusters, buses and latencies) as used by Nystrom et al., our basic algorithm produces schedules that have an IPC about 7% higher. In that paper, the proposed algorithm is evaluated with the configurations 2-cluster/2-buses and 4-cluster/4-buses (and both assuming 1-cycle latency buses). The results obtained there (even though for a set of programs different from ours) demonstrated that their scheduling algorithm obtained for 94% and 98% of the loops the same II as a unified machine with the same number of resources. We do not show our results in terms of II but in relative IPC, which is defined as the IPC obtained for the clustered configurations with respect to the unified configuration (this measure is more realistic since prolog, epilog and the actual number of iterations of each loop are taken into account). Looking at Figure 6.4, for the same configurations we can see that a strategy based on performing the cluster assignment and scheduling at the same time performs better than a scheme based on a two-step approach.

```
  // Compute scheduling for the original graph
(1)sched = ScheduleGrah(G);
  // Check if unroll is beneficious
(2)if (LimitedByBus(sched)) {
(3) ufactor = ncluster;
(4) comneeded = NDepsNotMult(G) * ufactor;
(5) cycneeded = (comneeded/nbuses) * latbus;
(6) if (cycneeded < II(sched)) {
(7)  G' = UnrollLoop(G, ufactor);
     return (ScheduleGraph(G'));
    }
  }
  return (sched);
```

Figure 6.5. Selective unrolling algorithm

The second important conclusion that we can draw from Figure 6.4 is that the performance of the clustered architecture significantly decreases when the number of buses decreases or the latency of the buses increases. This can be observed for both approaches although to a lesser extent for our proposal. This degradation is caused by the fact that the bus (or buses) becomes the bottleneck of the architecture.

6.3.3. Adding Loop Unrolling

As we have seen in the previous section, the communication buses may be the main performance bottleneck, even when the scheduling algorithm tries to reduce the number of communications among clusters. The alternative we propose to reduce the pressure on the buses is to apply the previous BSA scheduling algorithm to an unrolled graph. Loop unrolling is a well-known technique. Using both loop unrolling and modulo scheduling was proposed by Lavery and Hwu [62] in order to reduce resource requirements and the length of critical paths. Their observation was that using loop unrolling the actual *MII* (minimum initiation interval) for the unrolled loop is closer to the real *MII* when the value is rounded. In our case, the reason for applying loop unrolling is that many times loops present very few dependences among iterations (loop-carried dependences). Therefore, scheduling different iterations on different clusters require few communication and in addition, the workload is balanced since all iterations perform the same amount of work.

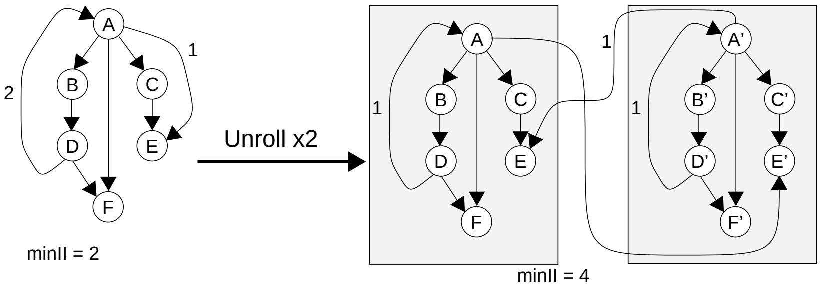
However, a drawback of loop unrolling is code expansion, which may be a critical issue in some systems such as embedded processors. Thus, it should be used only for those cases in which it provides a clear net benefit. For instance, if the performance of the non-unrolled loop is not limited by communications, unrolling may not provide any additional benefit. For this reason we propose an algorithm to perform loop unrolling only when it increases performance.

The selective unrolling algorithm is shown in Figure 6.5. First of all, the schedule of the graph without unrolling is computed. If the resulting schedule is limited by communications (i.e., the initiation interval was increased because the buses become saturated) then a schedule with the unrolled loop is tried. Our schedule algorithm presented in the previous section tends to schedule different iterations into different clusters. Therefore, the unroll factor is set to the number of clusters. Scheduling one iteration in each cluster results in a number of communications (comneeded) equal to the number of dependences at distance greater than zero (and not multiple of the unrolling factor) multiplied by the unrolling factor itself. Thus, the cycles needed to communicate the values (cycneeded) can be computed by dividing the total number of cycles needed for communications (comneeded * latbus) by the number of buses (nbuses). If this value does not increase the initiation interval of the unrolled loop (which can be determined without performing the scheduling), then the loop is finally unrolled and the scheduling of the new graph is performed.

An example of the scheduling process for a loop is shown in Figure 6.6. The resulting graph has two loop-carried dependences. The table in Figure 6.6 shows the scheduling process for the graph without unrolling. Suppose the architecture has two general-purpose functional units per cluster, each instruction is 1-cycle latency and one bus with one-cycle latency. The minimum *II* is computed as 2 (*ResMII* = $\lceil 6/4 \rceil$ = 2, and *RecMII* = $\lceil 3/2 \rceil$ = 2), and thus the maximum number of communications is 2. The nodes are scheduled following the computed order. In the table, *tmp* is the tmpoutedges value in our scheduling algorithm (see Section 5.1). We can see that nodes D, B, A and C are scheduled on cluster 0. However, node E and F cannot be scheduled in this cluster because it is already full (there are no free functional units). For node E, two communications are needed (values from A and C), and therefore the communication needed for F (value from D - value from A was previously brought) cannot be allocated. Therefore the *II* has to be increased to 3 in order to find a feasible scheduling. On the other hand, looking at the unrolled graph, the minimum *II* is 4 in this case, and thus 4 communications of 1 cycle are available. However, following our algorithm just 2 communications are needed (from A' to E and from A to E'), because different iterations are scheduled in different clusters. In this case, unrolling hides the communication latency (it would even if the latency of the bus was 2 cycles) and the unrolled schedule in more effective.

6.3.4. Evaluation

In this section we first show the different clustered VLIW configurations evaluated and list the set of benchmarks used to evaluate the performance of the scheduling algorithm. Then, some performance figures comparing unified and clustered architectures are shown including timing considerations. Finally, some results about the impact on code size of the unrolling technique are shown.



Nodes	CLUSTER 0		CLUSTER 1		CLUSTER CHOSEN	NCOMM	
	tmp	profit	tmp	profit			
D	1	1	1	1	0	0	✓
B	1	0	1	1	0	0	✓
A	2	1	1	1	0	0	✓
C	3	1	1	1	0	0	✓
E	-	-	0	0	1	2	✓
F	-	-	0	0	1	3	✘

Figure 6.6. Example of how to unroll a loop

Benchmarks and Configurations Evaluated

The scheduling algorithm has been evaluated for three different configurations of the VLIW architecture. This configurations are shown in Table 6.1.

RESOURCES	Unified	2-cluster	4-cluster
INT / cluster	4	2	1
FP / cluster	4	2	1
MEM / cluster	4	2	1
REGS / cluster	64	32	16

LATENCIES	INT	FP
MEM	2	2
ARITH	1	3
MUL	2	6
DIV/SQR/TRG	6	18

Table 6.1. Clustered VLIW configurations and latencies

The first configuration is called *Unified* and it is composed of a single cluster with four functional units of each type (integer, floating point and memory) and a unique register file of 64 general-purpose registers. This configuration represents our baseline. Both the *2-cluster* and *4-cluster* configurations have the register file partitioned (into two and four partitions respectively). The former has 2 functional units of each type and 32 register per cluster and the latter corresponds to 1 functional unit of each type and a register file of

16 registers per cluster (note that both, in total, are 12-way issue). For the clustered configurations we will show results for different number of buses (1 or 2) and with different latencies (1, 2, or 4 cycles).

For all configurations the memory hierarchy is shared by all the clusters and considered perfect (i.e., always hits with minimum latency). In the case of considering a real memory, techniques to reduce the impact of cache misses when modulo scheduling is applied should be used [90].

The modulo scheduling algorithm has been implemented in the ICTINEO compiler [4] and all the SPECfp95 benchmarks have been evaluated. The programs were run until completion using the test input data set. The performance figures shown in this section refer to the modulo scheduling of innermost loops with a number of iterations greater than four. We have measured that code inside such innermost loops represent about 95% of all the executed instructions, and then the statistics for innermost loops are quite representative of the whole program.

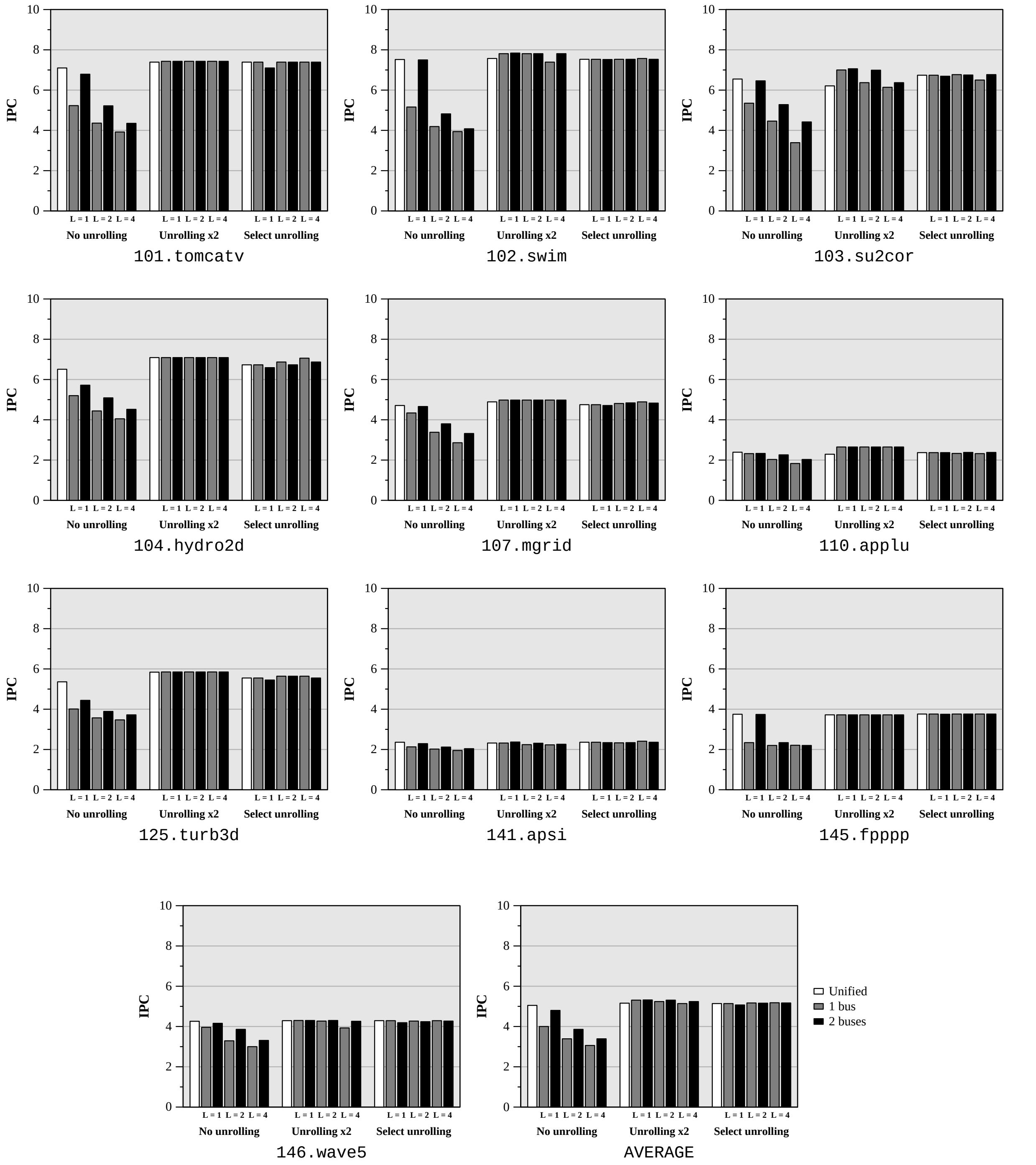
IPC Performance Figures

The results shown in this section show to the IPC (Instructions committed Per Cycle) obtained for the unified and clustered configurations for different values of the number of buses and latency. The IPC takes into account the prolog, the kernel and the epilog as well as the number of iterations and the times each loop is executed. Both non-unrolled and unrolled versions of the loops are evaluated.

The IPC results for all the SPECfp95 programs as well as average figures are shown in Figure 6.7 and Figure 6.8. Graphs on Figure 6.7 compare the *unified* configuration with the *2-cluster*, whereas graphs on Figure 6.8 compare the *unified* with the *4-cluster* configuration. Each graph is divided into three sets of bars:

- *No unrolling*: results when the loops are not unrolled.
- *Unrolling*: results when all the loops of the program have been unrolled. In the case of the 2-cluster configuration, the unroll factor is 2. In the case of the 4-cluster configuration this factor is 4.
- *Selective unrolling*: results using the selective unrolling algorithm presented in Section 6.3.3.

Each one of these sets if composed of different bars. White bars show the IPC obtained by the unified configuration. Grey bars show the IPC obtained by the clustered configuration with just 1 bus. Finally, black bars are the IPC achieved with clustered configurations and 2 buses. For clustered configurations, different latencies for the buses have been considered (L = 1, 2 or 4 cycles).



101.tomcatv

102.swim

103.su2cor

104.hydro2d

107.mgrid

110.applu

125.turb3d

141.apsi

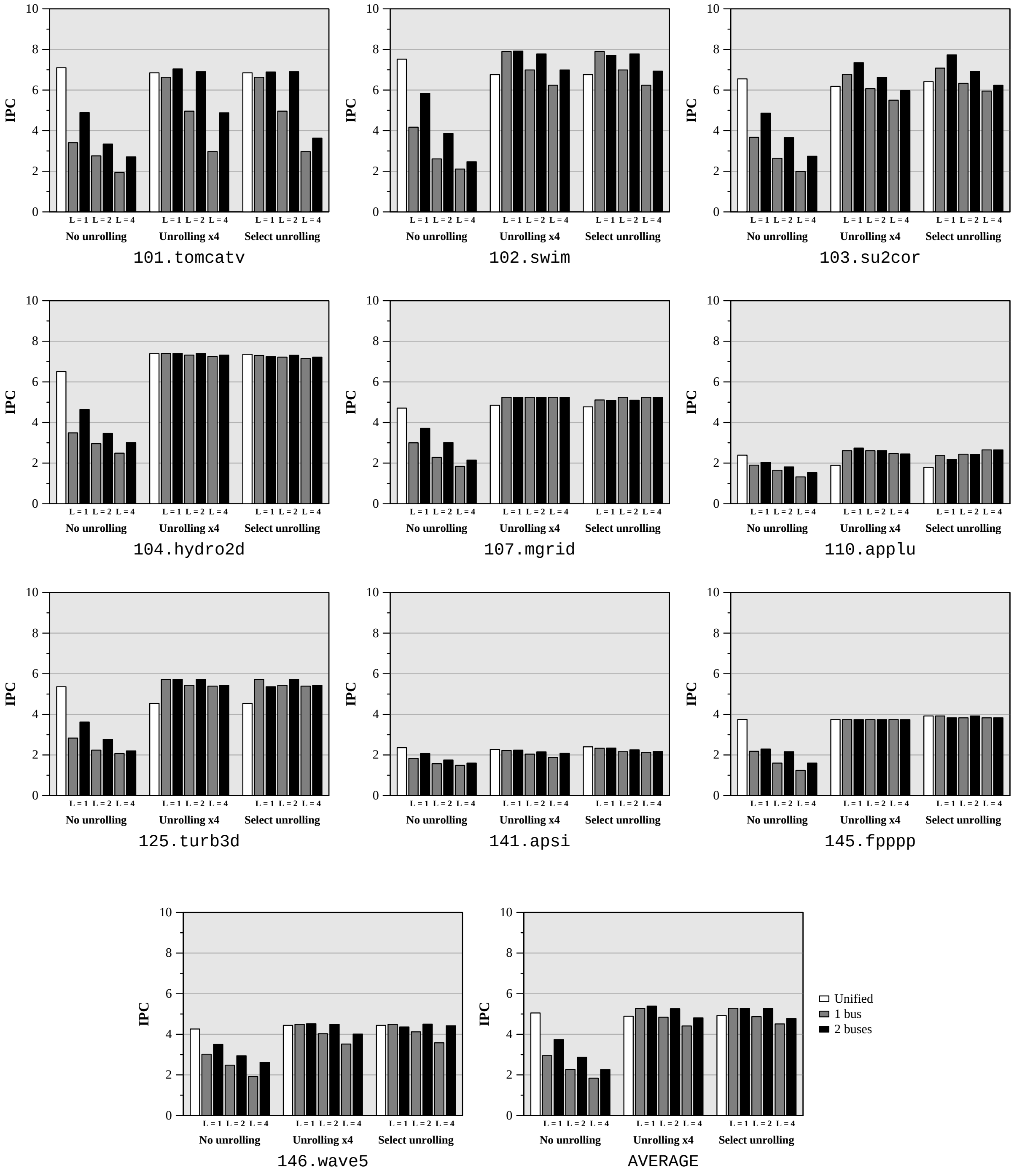
145.fpppp

146.wave5

AVERAGE

(a) 2-cluster configuration

Figure 6.7. IPC results for all the SPECfp95 benchmarks and a 2-cluster configuration



(b) 4-cluster configuration

Figure 6.8. IPC results for all the SPECfp95 benchmarks and a 4-cluster configuration

When we look at the first set of bars (*No unrolling*), we can see that the IPC achieved by clustered architectures compared with the unified architecture decreases when the number of buses decreases or the bus latency increases. We can see that this problem is overcome when loop unrolling is applied to all loops (*Unrolling*). The performance obtained for clustered architectures is the same (or even better) for most of the programs and configurations (except for `tomcatv` in the 4-cluster configuration). Note that when all loops are unrolled our scheduling algorithm is less sensitive to the number of buses and their latency. The reason why clustered architectures perform better than unified architectures for some programs and configurations when all loops are unrolled is due to our scheduling algorithm. When loop unrolling is applied, the different iterations of the loop are scheduled in different clusters, using their resources equally. However, in the unified architecture, all the resources are available when scheduling the first subgraph of the unrolled loop. As the scheduling phase tries to schedule operations as close as possible to their predecessors and successors in order to minimize register pressure, a very good scheduling is obtained for the subgraph of the first iteration sometimes at the expense of the other iterations.

The results for the selective unrolling presented in Section 6.3.3 are shown in the third set of bars (*Selective unrolling*). We can see that using this selective unrolling algorithm the performance obtained is very similar to the one obtained when all loops are unrolled. However, as we will see in Section 6.4, the code size is significantly reduced for this scheme.

Timing considerations

We have shown that the proposed scheduling algorithm applied to clustered architectures achieves about the same IPC as the unified configuration. However, the real benefit of clustered architectures comes when the cycle time is considered in the total performance. Using the delay models proposed by Palacharla [80], we show in Table 6.2 the cycle time (in picoseconds, for a technology of 0.18µm) obtained for the different configurations of the VLIW machine. In each case, we have assumed that the cycle time is determined by the maximum between the bypass delay and the access time to the register file. The former depends on the number of functional units per cluster, whereas the later depends on both the number of ports (2RD/1WR per functional units plus 1RD/1WR per bus) and the number of registers per cluster. Using the numbers of

Unified	2-cluster		4-cluster	
	1 bus	2 buses	1 bus	2 buses
1030.08 ps	394.12 ps	420.52 ps	293.69 ps	311.24 ps

Table 6.2. Cycle times according to Palacharla model

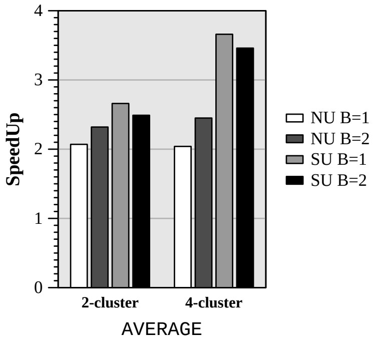


Figure 6.9. Speedup of clustered architectures with respect the unified one (bus latency=1 cycle)

this table, Figure 6.9 shows the average speed-up achieved by some clustered configurations with respect to the unified one. In this figure, NU stands for No Unrolling, whereas SU means Selective Unrolling. For both cases, there are results for one (B=1) and two (B=2) buses.

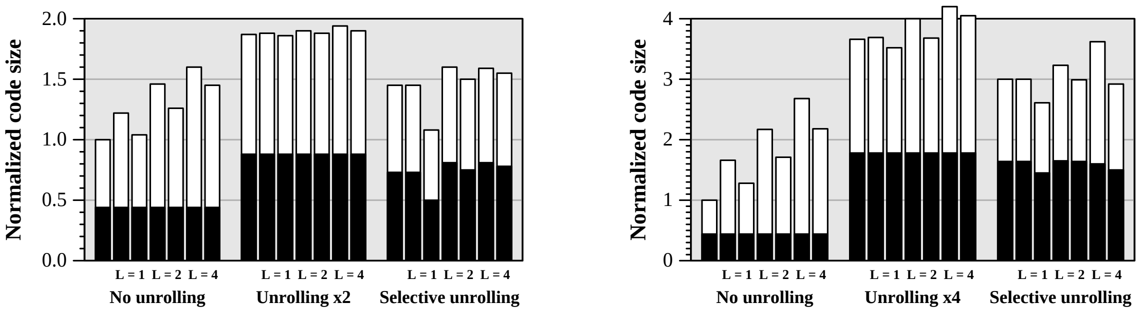
The main conclusion we can draw from this figure is that all configurations significantly outperform the unified configuration and the best performance is always obtained for the 4-cluster configuration with 1 bus when the selective unrolling algorithm is used, achieving an speed-up of 3.6 on average for the SPECfp95.

Effect on Code Size

Although loop unrolling is beneficial for modulo scheduled loops in a clustered VLIW architecture, code expansion in a major drawback of this technique. For those applications where code size in a major constraint, loop unrolling can bring another kind of problems (for instance, when code does not fit in the memory of an embedded processor). The selective unrolling proposed in Section 5.2 tries to unroll only those loops for which the bus is the main performance bottleneck.

The size of the code in a VLIW is a measure hard to obtain because compression techniques are commonly used. The compressed code size depends on the number of useful operations, the number of NOP operations and how they are distributed in the code. However, this topic is beyond the scope of this paper, and therefore we show just some measures in order to approximate the size of the code.

The effect of unrolling on the code size is shown in Figure 6.10. The different bars in the graphs correspond to the same scenarios as in Figure 6.7 and Figure 6.8. The graph on the left shows the results for the 2-cluster configuration, whereas the graph on the right is for the 4-cluster configuration. For each graph, each column in normalized to the size of the code for the unified configuration and without unrolling (first



AVERAGE
(a) 2-cluster configuration

AVERAGE
(b) 4-cluster configuration

Figure 6.10. Impact of loop unrolling in the code size

bar). White bars represent the amount of operations taking into account NOP operations, and black bars show just useful operations.

We can conclude from this figure that when loops are not unrolled, the number of NOP operations tends to increase when the latency increases or the number of buses decreases since the II augments. This trend does not appear when unrolling is performed. We can see that the selective unrolling algorithm decreases the total size of the code in terms of both useful and NOP operations. The decrement is better for configurations with higher communication bandwidth (i.e., 2 buses with 1-cycle latency).

6.4. Scheduling for a Fully-Distributed Architecture

In this section we adapt the Basic Scheduling Algorithm (BSA) presented in the previous section to a fully-distributed clustered VLIW architecture. The main characteristic of this new architecture is that, in addition to the functional units and the register file, the L1 cache is also partitioned among the clusters. Due to the resemblance of this novel architecture with a multiprocessor, we call this architecture as *MultiVLIW-Processor*. We first show the main features of this new organization. Afterwards, we show a motivating example that demonstrates the utility of incorporating a data locality analysis in the scheduler. Finally, the scheduling algorithm and performance results are presented.

6.4.1. Architecture

Our base architecture (see Figure 6.11) is composed of several clusters, each one executing a fixed part of each VLIW instruction. All clusters work in lockstep mode, i.e., any stall in one cluster also stalls the other clusters. Every cycle, all clusters fetch their corresponding parts of a new VLIW instruction from their local instruction caches. Each cluster consists of several functional units, a register file and a local data cache memory in addition to the local instruction cache. Functional units can be of three different types:

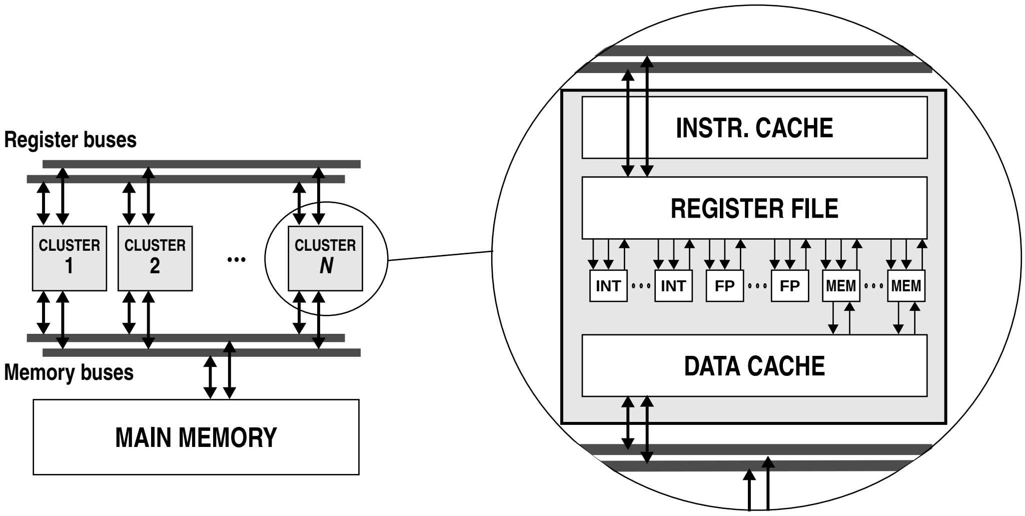


Figure 6.11. Microarchitectures of a MultiVLIWProcessor

integer arithmetic, floating-point arithmetic or memory access. For the sake of simplicity, we consider that all clusters are homogeneous (i.e., with the same number and type of functional units), but the proposed techniques can be generalized for heterogeneous clusters. The format of the VLIW instruction and the data paths are the same as in Figure 6.1 and Figure 6.2.

Regarding memory accesses, a load/store issued by a cluster first tries its local L1 data cache. If the data is found, the access is satisfied with minimum latency. Otherwise, the cache of the other clusters are searched or, finally, the access is solved by the main memory. Both local memories and main memory are interconnected through one or several buses (that are called *memory buses*). As the cache is physically partitioned among the clusters, coherence among the local caches and the main memory has to be kept. For this reason, a snoopy MSI protocol [13] has been assumed. This protocol is completely transparent to the ISA, and further, both the coherence and the bus arbitration are managed by the hardware. When a memory access misses in its local cache, the miss request is queued in a local MSHR (*Miss information/Status Handling Register*) structure, since the L1 data cache is non-blocking [60]. Then, the access has to compete for a free memory bus in order to access a remote cache or the main memory.

All the dependences with memory operations are dynamically checked, since the scheduler may have considered an optimistic latency for these instructions (i.e., hit in the local cache). If any dependence is not met, the dependent instruction stalls in all clusters until the hazard is resolved.

6.4.2. Motivation

The two main parameters that statically characterize a modulo scheduled loop are the *initiation interval* (II) and the *stage count* (SC), and for a clustered VLIW architecture both of them can be affected by inter-cluster register communications. For this section, which focuses on modulo scheduling for *multiVLIWprocessors*, the number of cycles needed to execute a particular modulo scheduled loop can be modeled through the following expression (as previously seen in Chapter 5):

$$\mathrm{NCYCLE}_{\mathrm{Total}} = \mathrm{NCYCLE}_{\mathrm{Compute}} + \mathrm{NCYCLE}_{\mathrm{Stall}}$$

Where $\mathrm{NCYCLE}_{\mathrm{Compute}}$ represents a fixed number of cycles that depends on the particular static scheduling produced by the compiler. During these cycles the processor is doing useful (or at least scheduled) work. $\mathrm{NCYCLE}_{\mathrm{Stall}}$ represents the number of cycles where the processor is stalled and depends on several factors as we detail below. The value of $\mathrm{NCYCLE}_{\mathrm{Compute}}$ can be computed before executing the loop if the number of times the loop is executed (NTIMES) and the number of iterations of each execution (NITER) are known, as shown by the next expression:

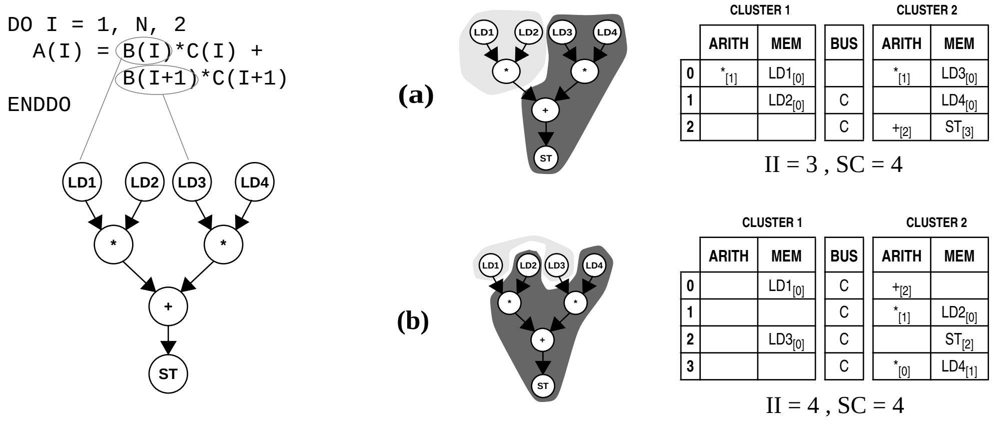
$$\mathrm{NCYCLE}_{\mathrm{Compute}} = \mathrm{NTIMES} * ((\mathrm{NITER} + \mathrm{SC} -1) * \mathrm{II})$$

The value of $\mathrm{NCYCLE}_{\mathrm{Stall}}$ cannot be computed statically. It represents the number of stall cycles due to incomplete information managed by the compiler. For instance, some memory instruction latencies may be unknown since the compiler does not know whether they will hit in the first level cache. If the value loaded by a memory instruction feeds another operation (i.e., the latter depends on the former) but the latter was scheduled using an underestimation of the memory latency, it will stall until the memory access is finished. In the assumed microarchitecture, the final latency of a memory instruction depends on three factors:

- Latency of memory accesses, which depends on the memory level that satisfies the access: local cache, remote cache or main memory.
- Number of entries in the MSHR of the lockup-free caches. If there is no available entry for a new miss request, the instruction stalls until there is a free entry.
- Cycles waiting for a free bus and bus latency.

Thus, considering all of these factors, the total latency of a memory access can be represented by this formula:

$$\mathrm{LAT}_{\mathrm{MemAccess}} = \mathrm{LAT}_{\mathrm{Cache}} + \mathrm{MISS}_{\mathrm{LC}} * (\mathrm{NC}_{\mathrm{WaitingEntry}} + \mathrm{NC}_{\mathrm{WaitingBus}} + \mathrm{LAT}_{\mathrm{MemoryBus}} + \max (\mathrm{LAT}_{\mathrm{Cache}} , \mathrm{MISS}_{\mathrm{RC}} * \mathrm{LAT}_{\mathrm{MainMemory}}))$$



```
DO I = 1, N, 2
  A(I) = B(I)*C(I) +
         B(I+1)*C(I+1)
ENDDO
```

(a)

	CLUSTER 1			CLUSTER 2	
	ARITH	MEM	BUS	ARITH	MEM
0	$*_{[1]}$	LD1$_{[0]}$		$*_{[1]}$	LD3$_{[0]}$
1		LD2$_{[0]}$	C		LD4$_{[0]}$
2			C	$+_{[2]}$	ST$_{[3]}$

II = 3 , SC = 4

(b)

	CLUSTER 1			CLUSTER 2	
	ARITH	MEM	BUS	ARITH	MEM
0		LD1$_{[0]}$	C	$+_{[2]}$	
1			C	$*_{[1]}$	LD2$_{[0]}$
2		LD3$_{[0]}$	C		ST$_{[2]}$
3			C	$*_{[0]}$	LD4$_{[1]}$

II = 4 , SC = 4

Figure 6.12. Motivating example

Where both MISS_{LC} and MISS_{RC} represent binary values that are 1 if the access misses in local cache and all remote caches respectively, or 0 otherwise. $\text{NC}_{\text{WaitingEntry}}$ represents the number of cycles that a miss access is waiting for an available entry in the MSHR. $\text{NC}_{\text{WaitingBus}}$ is the number of cycles that the access is waiting for a free bus. Note that a bus can be also busy for coherence operations and this is taken into account by our simulator. Finally, although we have considered $\text{LAT}_{\text{MainMemory}}$ as a fixed parameter, in the above expression note that for some references this number could be smaller if an earlier miss has already started loading the relevant cache line. This fact has also been accounted for by our simulator.

Motivating Example for the Proposed Scheduler

The objective of this study is twofold: first, demonstrate that when the data cache is partitioned among the different clusters, the selection of the cluster where each memory instruction is scheduled is very important and can dramatically affect the final performance of a program (the same holds for register values, but this has already been shown in Section 6.3). Second, we propose a modulo scheduler that takes into account both register and memory inter-cluster communications.

In this section, we illustrate through an example how the cluster selection can affect the total number of cycles in which a code section is executed. Consider that we want to perform modulo scheduling of a loop whose code and dependence graph are shown in Figure 6.12. Assume the processor consists of 2 clusters, each one with its local register file and data cache (direct-mapped), and 2 functional units: one for arithmetic operations (with 2-cycle latency) and one for memory operations. There is one inter-register bus with a 2-cycle latency. The latencies for memory accesses are: 2 cycles for a local cache, 2 cycles for a bus transaction and 10 cycles for an access to main memory.

For this loop, the *minimum initiation interval* (mII) for an equivalent unified architecture with the same resources is 3 cycles. The partition and scheduling that minimizes the number of register communications between clusters and, thus, that achieves the same II as the equivalent unified architecture is shown in Figure 6.12(a). In this figure, the left part represents the partition of the operations between the clusters whereas the right part shows the modulo reservation table obtained after modulo scheduling. Each operation is scheduled in a particular slot and the number in brackets represents the stage at which this operation is scheduled. The usage of the register bus is also shown in this table. Whenever a bus transaction takes place, the corresponding bus time slot is reserved and it is indicated by a *C* in the reservation table.

Then, the $\text{NCYCLE}_{\text{Compute}}$ of the resulting loop can be computed as:

$$\text{NCYCLE}_{\text{Compute(a)}} = \text{NTIMES} * ((N + 4 - 1) * 3) = \text{NTIMES} * (N + 3) * 3$$

However, suppose that both arrays `B` and `C` are located in memory at a distance that is a multiple of the local cache memory size. This means that we will have ping-pong interferences between LD1 and LD2, and between LD3 and LD4. Thus, the spatial locality exhibited by the four instructions cannot be exploited and the four accesses always miss. The result is that the instruction(s) that consume the memory values suffer many stalls. In the example, the VLIW instruction that contains the multiplications cannot continue its execution until the misses are satisfied. Assuming that we have sufficient memory buses, the number of cycles that the instruction stalls is the latency of a bus transaction plus an access to main memory, since the latency to the local cache was taken into account by the scheduler. Then, the number of stall cycles is:

$$\text{NCYCLE}_{\text{Stall(a)}} = \text{NTIMES} * N * (2+10) = \text{NTIMES} * N * 12$$

An alternative scheduling is shown in Figure 6.12(b). Based on the locality properties previously observed, in this second alternative cluster assignment is selected in order to take advantage of the locality exhibited by memory instructions. For this reason, LD1 and LD3 are scheduled in the same cluster in order to benefit from its group reuse, and the same applies for LD2 and LD4 which are scheduled in the other cluster. In this way, ping-pong interferences are removed and we can take advantage of the spatial reuse. However, as we can see in the example, for this case two communications between register values are needed per iteration, and then the II has to be increased from 3 to 4. Thus, $\text{NCYCLE}_{\text{Compute}}$ is computed as:

$$\text{NCYCLE}_{\text{Compute(b)}} = \text{NTIMES} * ((N + 3 - 1) * 4) = \text{NTIMES} * (N + 2) * 4$$

However, the miss rate of LD3 and LD4 is 25% (assuming eight data elements per cache block), and LD1 and LD2 always hit (excepting the first iteration). Thus, the number of stall cycles is:

$$\text{NCYCLE}_{\text{Stall(b)}} = \text{NTIMES} * \text{N} * (2*(2+10)* 0.25) = \text{NTIMES} * \text{N} * 6$$

Then, putting all together, we have that the total number of cycles in both strategies as:

$$\text{NCYCLE}_{\text{Total(a)}} = \text{NTIMES} * (15 * \text{N} + 9)$$

$$\text{NCYCLE}_{\text{Total(b)}} = \text{NTIMES} * (10 * \text{N} + 8)$$

Therefore, we can conclude that the second strategy, which takes into account both register and memory communications, achieves a schedule that is 1.5 times faster than the original one, which is optimized only for register communications.

6.4.3. Scheduling Algorithm

In this section we present a modulo scheduler that tries to minimize both register and memory inter-cluster communications and at the same time balance the workload. We will compare the performance of the proposed algorithm with the results obtained for the *Basic Scheduling Algorithm* (BSA), which ignores the effect of cache misses.

The new algorithm makes use of a data locality analysis when perform the instruction scheduling. So that, we use the CME as proposed in Chapter 2 to estimate the amount of reuse that is exploited by any subset of memory instructions. CME will allow the scheduler to estimate the amount of memory communications among clusters or between clusters and main memory. The scheduler uses this information to guide its scheduling decisions. For instance, given a memory instruction, it is beneficial to schedule it in a cluster where there already are other instructions from which it reuses data (group reuse). On the other hand, it is detrimental to schedule the instruction in a cluster where there already are other instructions that cause many cache conflicts with the current one. CME allow the schedule to quantify the amount of reuse and conflicts among any group of instructions of the same loop nest. CME are used to produce the following statistics:

- The number of misses incurred by a set of memory references for a particular cache configuration (capacity, block size and associativity)
- The miss ratio of a particular memory instruction in this set.

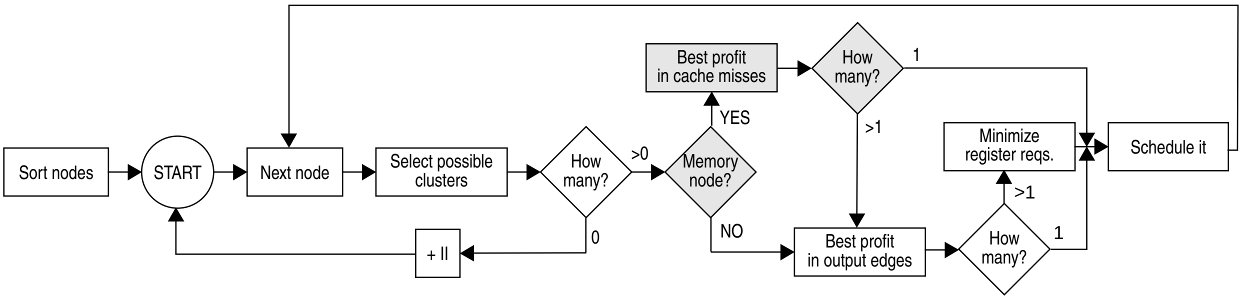


Figure 6.13. RMCA modulo scheduling step by step

Scheduler for a Distributed Cache

The proposed algorithm is called *RMCA* (which stands for *Register and Memory Communication-Aware*) modulo scheduling. It is an evolution of the BSA algorithm presented in Section 6.3.2 and its main steps are depicted in Figure 6.13 (new features are shown in gray boxes). All nodes in the data dependence graph are first sorted according to the criteria used by the BSA algorithm. This ordering minimizes the number of nodes that have both predecessors and successors in the set of nodes that precede it in the order. Then, cluster selection and scheduling is performed in a single step following that order. However, there is now a distinction between two types of nodes: (a) memory operations, and (b) non-memory operations. For operations of the latter group, the algorithm proceeds as the BSA scheme. However, when a memory operation is scheduled, a different strategy is used. Instead of choosing the cluster where the gain from output register edges is maximized, the cluster selection depends on the profit from cache misses. In other words, each time a memory operation is scheduled, all clusters are tried, and for each one, the number of cache misses contributed by memory operations scheduled in that cluster, before and after introducing the current operation, is computed through the CME. Then, the cluster(s) where this gain is maximized is chosen. If more than one cluster is optimal with respect to cache misses, the scheduler selects one of them using the same strategy as for non-memory operations. Although the solver of the CME have to be repeatedly invoked, the method is very fast due to the optimizations mentioned in Chapter 2., and the time required by the scheduler is a small percentage of the total compilation time.

This algorithm tries to minimize the number of cache misses, and thus it attempts to minimize the inter-cluster memory communications. However, the latency of these communications can be hidden by scheduling some load instructions using the cache-miss latency (binding prefetching, as proposed in Chapter 5). When a load is scheduled using the cache-miss latency, the operation that consumes the data read by the load will not be stalled because it is scheduled assuming the worst-case latency. However, scheduling instructions using a larger latency can have a negative effect on both register pressure and length of the schedule. On one hand, the lifetime of the load destination register is increased. On the other hand, the II can be increased if this instruction belongs to a recurrence and this increased latency makes the recurrence the most restrictive constraint on the II. Besides, the length of the schedule for a single iteration may increase, which may cause an increase in the SC, which in turn affects the durations of the prolog and epilog. Therefore, as shown in Section 6.4.2, it may be much more effective to schedule with a miss latency only those loads that are likely to miss. This can be done as long as the latency does not increase the II with respect to the schedule produced when loads are scheduled with a hit latency. Thus, the proposed scheme includes another step: once the target cluster of an instruction is determined, it is scheduled using the cache-miss latency if the miss ratio of this instruction in this particular cluster (considering the partial schedule produced so far) is greater than a certain threshold, and provided that this latency does not increase the II if the operation is in a recurrence. The assumed miss latency is the time to access main memory, that is, $\mathrm{LAT}_{\mathrm{Cache}} + \mathrm{LAT}_{\mathrm{MemoryBus}} + \mathrm{LAT}_{\mathrm{MainMemory}}$ (note that we do not consider the memory bus contention since it is not known at this moment, although it could be estimated).

Note that with this scheme some memory instructions are scheduled with the miss latency even if their miss ratio is lower than 100%. This may happen for instance for instructions with spatial locality. In this case, loop unrolling could be used to generate multiple instances of the same instruction such that one of them always miss and the other always hit (as shown in Chapter 5). However, we have not considered this optimization in this paper.

6.4.4. Evaluation

This section analyzes the performance of the proposed scheduler. The main performance metric that we use is the number of cycles executing instructions of modulo scheduled loops. Note that this metric does not include the effect of clustering on the cycle time, thus, differences observed for different schedulers and the same architecture directly translate into differences in execution time. However, the number of cycles for different architectures should be divided by cycle time to measure differences in execution time. Since we are concerned with differences among alternative schedulers, we prefer not to include the effect of cycle time in our metric, to isolate the effect of the schedulers. A study of the impact of clustering on cycle time can be found elsewhere [80] as well as on energy consumption [113], which is another important factor that can be reduced through clustering.

Configurations and Benchmarks

The scheduling algorithm has been evaluated for three different configurations of the *multiVLIWprocessor* architecture. These configurations are shown in Table 6.3. The first configuration is called *Unified* and it is composed of a single cluster with four functional units of each type (integer, floating point and memory) and a unique register file of 64 general-purpose registers. This configuration represents our baseline.

RESOURCES	Unified	2-cluster	4-cluster
INT / cluster	4	2	1
FP / cluster	4	2	1
MEM / cluster	4	2	1
REGS / cluster	64	32	16

LATENCIES	INT	FP
MEM	2	2
ARITH	1	3
MUL	2	6
DIV/SQR/TRG	6	18

Table 6.3. Clustered VLIW configurations and latencies

Both the *2-cluster* and *4-cluster* configurations have the register file partitioned (into two and four partitions respectively). The former has 2 functional units of each type and 32 register per cluster and the latter includes 1 functional unit of each type and a register file of 16 registers per cluster. The three configurations are 12-way issue.

For all configurations, the total L1 cache size is 8KB, divided into equal-sizes among the different clusters. This cache capacity is realistic for embedded/DSP processors. For instance, the TI TMS320C6711 has an L1 data cache of 4Kbytes [101]. In our architecture, each local cache is direct-mapped, non-blocking with 10 entries in the MSHR. An access to a local cache is satisfied in 2 cycles, whereas an access to main memory takes 10 cycles. For the clustered configurations we will present results for different number and latency of both register and memory buses.

The modulo scheduling algorithm has been implemented in the ICTINEO compiler [4] and some SPECfp95 benchmarks have been evaluated: *tomcatv*, *swim*, *su2cor*, *hydro2d*, *mgrid*, *applu*, *turb3d* and *apsi*. Note that modulo scheduling is an effective technique for both numeric and multimedia applications, but it is not so effective for applications such as SPECint95 due to the small number of iterations for each loop execution and the abundance of conditionals.

The performance figures shown in this section refer to the modulo scheduling of innermost loops with a number of iterations greater than four. Our measurement shows that code inside such innermost loops represents about 90% of all the executed instructions, so that the statistics for innermost loops are quite representative of the whole program. Only instructions that belong to modulo scheduled loops are taken into account by the simulator. Thus, the programs were run until the first 100 million memory instructions in these loops using the ref input data set.

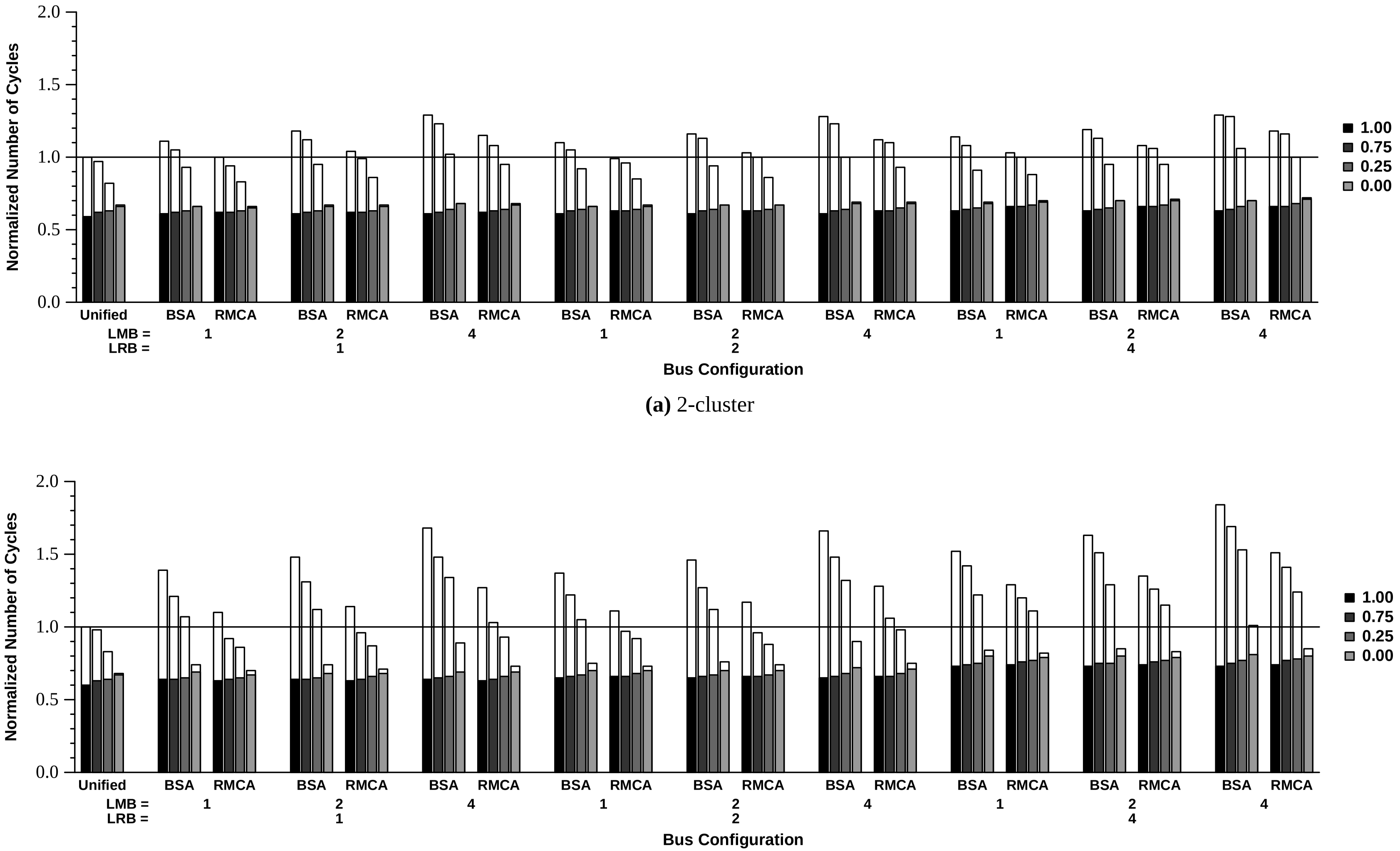
An Unbounded Number of Buses

Before considering realistic configurations, we have evaluated an architecture with an unbounded number of buses to test the performance of the proposed algorithm under extreme situations where bus bandwidth in not a problem. The remaining parameters of the architecture are those listed in Table 6.3 and the latency of the buses is parametrized. Figure 6.14 shows the normalized number of cycles averaged for all benchmarks, for 2 and 4 clusters and the different latencies considered. The first set of four bars represents the results for the unified configuration. The rest represent the results for the clustered configuration for different latencies of register buses (LRB - *Latency of Register Buses*) and memory buses (LMB - *Latency of Memory Buses*). For the different sets, we have evaluated two different schedulers:

- The BSA scheduler described in Section 6.3.2, which is very effective at minimizing register communications.
- The proposed algorithm, that takes into account both register and memory communications, which is labeled as *RMCA*.

Each set of four bars represents the results obtained for different values of the cache miss threshold (from 1.00 to 0.00) that determines whether a load is attempted to be scheduled with a miss latency. Note that threshold 1.00 represents the traditional scheme, that is, using always the cache-hit latency for memory operations. On the other hand, threshold 0.00 is most similar to the one proposed in Chapter 5, where all operations that do not cause an increment in the II (due to recurrences) are scheduled using the cache-miss latency. The only difference is the locality analysis employed, which is more powerful in this paper. Each bar is split into two parts: the compute time (or $\text{NCYCLE}_{\text{Compute}}$) is the black/grey part, whereas the stall time (or $\text{NCYCLE}_{\text{Stall}}$) is the white one.

From these graphs we can see that for all configurations (number of clusters, latencies and thresholds) the scheme that takes into account memory communication (*RMCA*) outperforms the one that ignores this feature (*BSA*). As expected, for smaller values of the threshold the compute time increases (since it may increase both the II due to register requirements, and the SC due to an increase in the length of the schedule) but the stall time decreases. Note that with a threshold of 0.00 the stall time is almost zero for all configurations and the number of cycles for the *multiVLIWprocessor* are comparable to those of the unified configuration. We can also observe that for small thresholds (0.25 or 0.00) both *BSA* and *RMCA* strategies



(a) 2-cluster

(b) 4-cluster

Figure 6.14. Results obtained for an unbounded number of buses (averaged for all benchmarks)

achieve similar performance, since the latency of cache misses is hidden by scheduling loads with the cache-miss latency, except for the 4-cluster configuration and memory bus latency of 4 cycles. Nevertheless, note that for an unbounded number of buses the time waiting for a free bus ($\mathrm{NC}_{\mathrm{WaitingBus}}$) is zero, and hence, if the latency is hidden, the number of misses has no effect. However, as we will see in next section, when the number of memory buses is limited, the difference between both schemes will be notable, since the schedules produced by the *RMCA* scheme require much less communications.

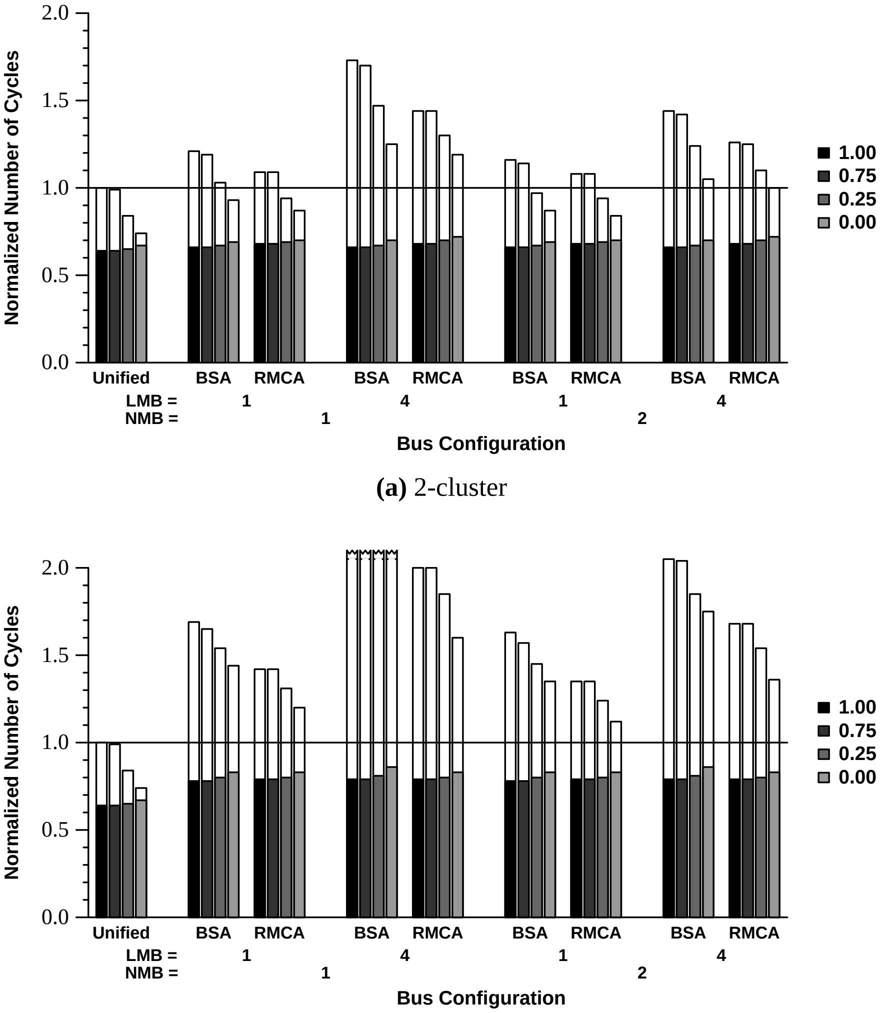
Evaluation of Realistic Configurations

We have shown the potential benefits that can be achieved when memory communication are taken into account by the scheduler. In this section we study the results when a realistic inter-cluster communication network is considered.

We have evaluated configurations with a fixed number and latency of register buses (2 buses with 1-cycle latency) and for a different number and latency of memory buses. In Figure 6.15 we can see the results for both 2 and 4 clusters. Each set of four bars has the same meaning as in the previous section. The first set represents the results for the unified configuration. The rest are the averaged results for the different strategies (*BSA* and *RMCA*) for 1 and 2 buses (NMB - *Number of Memory Buses*) and 1 and 4 cycles of latency (LMB - *Latency of Memory Buses*). We can observe in these graphs that, as in the unbounded study, the *RMCA* strategy outperforms the *BSA* for all configurations. However now, for small values of the threshold, the difference between both strategies is more remarkable, mainly for 4 clusters. For the most effective threshold (0.00), the *RMCA* scheme outperforms the baseline scheduler by about 5% for 2 clusters and 20% for 4 clusters. We have observed that the reason for this difference is the time spent waiting for an available bus in order to initiate a communication. When the number of memory buses is unbounded this value is zero, because there is always an available bus. However, when the number of buses is limited, reducing the number of misses is also important since lesser the number local cache misses, lesser the number of accesses competing for a free bus time slot.

6.5. CHAPTER SUMMARY

We have presented an effective approach to perform modulo scheduling for a clustered VLIW architecture. We have first proposed an algorithm oriented to reduce inter-cluster register communications. The performance of the proposed technique comes from using a single step to perform cluster assignment and instruction scheduling as well as from the use of a selective loop unrolling. We have shown that the resulting algorithm is very effective for a variety of configurations with different communication latency and



(a) 2-cluster

(b) 4-cluster

Figure 6.15. Results obtained when the number of buses is limited (averaged for all benchmarks)

bandwidth. Besides, the selective unrolling policy minimizes the impact of unrolling on the code size. Performance evaluation for the SPECfp95 shows that the IPC of the clustered architecture is not degraded in comparison with a unified architecture with the same resources. Moreover, when the cycle time of each architecture is considered, we have shown that a 4-cluster architecture is on average 3.6 times faster than a unified configuration.

In the second part of the chapter we have proposed a novel microarchitecture called *multiVLIWprocessor*, which has a fully-distributed clustered VLIW organization. The main novelty of this architecture with respect to previous proposals for clustered VLIW processors is the distributed data cache, which introduces new challenges to the instruction scheduler. Then, we have presented a modulo scheduler designed for this particular architecture based on the previous scheduler. The new scheduler, by means of a powerful locality analysis based on the *Cache Miss Equations* and an analysis of the register data dependence graph, generates codes with very low inter-cluster communication requirements. We have also shown that the proposed scheduler outperforms previous schemes that just focused on register communications.

7

CONCLUSIONS AND FUTURE WORK

This chapter presents the main conclusions and the open research lines of this work.

7.1. CONCLUSIONS

In this thesis we have presented three novel techniques to improve the performance of the first level cache that strongly rely on a powerful locality analysis. A main feature of the proposed techniques is that make use of some hints in the instructions that are statically set by the compiler using an extra pass that performs the locality analysis.

In Chapter 2 we have presented the two data locality analyzers that have been used in this thesis. The first one, called SPLAT, has been developed in this work. The analysis is divided into two steps. The first one collects static information (such as the reuse vectors or the loop nest organization) and profiling information (though a simple profiler of basic blocks). This information is the input of the second step, that is composed of three phases: reuse, volume and interference. The reuse and volume phases are common in other techniques proposed so far, however the interference phases is a novel contribution. We have checked the accuracy of SPLAT by comparing its results with the output of a cache simulator. We have seen that the tool is very accurate for the programs studied. However, the main advantage with respect the simulators is that it is much faster (it takes several seconds for most SPECfp95) and the locality information that can collect is very diverse. This makes the tool very flexible but also fast and precise.

The second locality analysis used in this thesis is the FastCME, developed by other authors. This analyzer uses mathematical and statistical techniques to solve the Cache Miss Equations. The result is a tool which is also fast, flexible and even more accurate than the SPLAT. A main advantage with respect to SPLAT is that with FastCME we can analyze the locality behavior of a program for set-associative caches.

In Chapter 3 we have presented some different locality statistics obtained using SPLAT. These statistics range from the study of the intrinsic reuse of an application, to the isolation of different critical sections of a code that are responsible for the majority of cache misses. The main conclusions that are extracted from this chapter are that different programs present very different reuse: whereas in some cases the temporal reuse is predominant, in some other cases is the spatial or both together. Moreover, this is also true for individual instructions or for code sections. The reason of why cache misses occurs also varies in single memory instructions, loops or the whole program. In some cases there is a predominant kind of miss (compulsory, capacity or conflict) whereas in some other cases there is a mixture of two or the three types. This study has motivated optimizations targeted to the particular features of each memory instruction. In this analysis we have also shown that the only-temporal reuse and only-spatial reuse are very common in some cases, which has motivated the cache organization proposed in Chapter 4. Summarizing, the most important conclusion drawn from this chapter is that locality behavior of memory instructions is statically predictable for the studied programs and then this information can be used to perform a smarter management of the cache.

In Chapter 4 we have used the locality analysis to statically set hints in memory instructions to manage the Selective Data Cache. This cache allows some references to bypass the cache in case of a miss. We have adapted the SPLAT to set a bit in each memory instruction to select whether it bypasses the cache or not. Results show improvements in miss ratio (since many interferences are eliminated) and traffic with the next level of the memory hierarchy. The effectiveness of the locality analysis to manage the Selective Data Cache and the statistics obtained in Chapter 3 have motivated the proposed LSMCache (Locality Sensitive Multimodule Cache). This cache is composed of three modules: one configured to exploit temporal reuse, one configured to exploit spatial reuse and a third one oriented to exploit both types of locality. In this case, the compiler is responsible to set the hints in memory instructions that indicate, on a miss, where a new block has to be stored. Results have shown the effectiveness of this architecture: compared with a conventional cache with equivalent capacity, the performance is much better. Moreover, if the spatial cache is configured with longer lines, the miss ratio obtained is near the miss ratio of a 64KB fully-associative cache (used as a lower bound). For this module we have also studied the effectiveness of including a simple hardware prefetching mechanism. We have evaluated different alternatives and the conclusion is that adding one-block lookahead prefetching to the LSMCache the performance is much better than the miss ratio obtained for the 64KB fully-associative cache. This study has shown that dealing each memory instruction in the most convenient way depending on its locality can significantly improve the performance.

In Chapter 5 we have studied the interaction between software pipelining and software prefetching. Whereas the main goal of software pipelining is to increase the ILP in a loop, software prefetching is oriented to hide memory latency. We have shown that modulo scheduling schemes using cache-hit latency for loads produce many stalls due to dependences with memory instructions. For a simple architecture the stall time represents about 32% of the execution time and 63% for an aggressive architecture. This mean that modulo schedulers that ignore the effect of stalls due to cache misses can obtain results far away from the ideal execution time. Then, we have compared two different approaches to introduce software prefetching: one based on early scheduling of memory instructions (also called binding prefetching) and another that inserts prefetch instructions (also called non-binding prefetching). Both of them significantly improve the performance, but they can cause significant penalties in some cases. In average, schemes based on early scheduling produce better results. For this reason we have proposed an heuristic scheduling algorithm (called CSMS - Cache Sensitive Modulo Scheduling), which is based on early scheduling and tries to minimize both the compute and the stall time. The proposed algorithm schedules some memory operations using the cache-miss latency using information about the shape of the dependence graph and information about the locality of each memory instruction obtained with the SPLAT. The results have shown that it outperforms the rest of strategies. Compared with a baseline scheme, which schedules all memory operations using the cache-hit latency, the produced code is 1.6 times faster for a simple architecture and 2.5 for an aggressive architecture.

Finally, Chapter 6 is focused on the schedule of instructions in clustered VLIW architecture, and in particular in schemes based on modulo scheduling. First we have studied how to perform modulo scheduling in a semi-distributed clustered architecture. The clusters in this architecture contain a local register file and functional units, but the cache (and memory hierarchy in general) is shared by all the clusters. The basic idea has been to propose a good algorithm to reduce communications among register files. We have proposed an algorithm whose main difference with previous works is that both the cluster assignment and instruction scheduling are performed into a single step instead of two sequential steps. We have shown that this methodology improves performance. Moreover, for loops where the performance is constrained by communications among clusters, we have studied the effect of loop unrolling. We have seen that including a selective loop unrolling in the algorithm is straightforward and very effective. Then, we have studied some heuristics to schedule instructions in fully-distributed clustered architectures. The basic difference with previous architecture is that now the cache is also distributed among the different clusters. Using as a base the previous scheduling algorithm and the information given by the FastCME tool, we have proposed an scheme (called RMCA - Register and Memory Communication Aware modulo scheduling) that takes into account both inter-cluster register and memory communications. Results have shown that the selection of the cluster where memory operations are scheduled is a key factor for performance.

7.2. FUTURE WORK

In this section the main research activities that we consider worthy to purse as a continuation of this thesis are summarized. The different ideas are split in the main topics that have been studied in this thesis. A general research task is to study the impact of the selected locality analysis in the performance of the proposed techniques. In this thesis we have used two possible data locality analyzers (SPLAT and FastCME), but we have not compared the results obtained when both are used for the same technique. It looks clear the better the locality analysis is, the better the propsed techniques perform, but we have not checked how must the accuracy of the locality analysis affects the results.

Data locality analysis

There are two points in which we are interested. First, we plan to study of reuse between loops. Both data locality analyzers presented in Chapter 2 were based on the reuse vectors. These vectors only represent the reuse among memory instructions in a loop, but not among instructions that belong to different loops. Some studies point out that this kind of reuse can be significant in some programs.

The second issue is a locality analysis for non-numerical codes. The work developed in this thesis is oriented towards numerical applications. The main reason for that is another limitation of the reuse vectors: they can only deal with affine references. For the SPECfp95 programs, that are the benchmarks used in this thesis, the majority of references follow this pattern. However, and mainly for non-numerical applications usually written in C language, many memory references are not affine. For instance, the use of pointers is very common. We plan to study the possibility of obtaining the reuse vectors for these kind of references using a different approach based on the study of how pointers are modified in a program. However, an alternative option is the use of profiling information or even a locality analysis implemented in hardware.

Study of locality behavior

Another utility of the locality analysis that has not been studied in this thesis is the use of the locality results to guide different optimizations. We have shown that different programs exhibit very different locality characteristics. Detailed evaluation of the locality exhibited by a program may then be essential to choose the best approach to improve it. Fully-automatic optimization tools have proved so far insufficient due to the variety of different scenarios that they should cope with. We then believe that the best approach today towards memory optimization is by means of an iterative (and interactive) process in which repetitive analysis and optimization steps are interleaved until the final result is acceptable. Therefore, the speed of the analysis tool as well as the range of information that it can provide are critical. We have shown that the type of analysis presented in this thesis can be very useful for such an approach.

Management of locality sensitive caches

About this topic there are three different lines of study. In this thesis we have shown the effectiveness of a multimodule cache with explicit software management. The results have been obtained feeding a cache simulator with memory traces. The first improvement in the performance evaluation would be to include the cache organization in a timing simulator. The main reason for that is that in such simulator we could take into account things such as bus contention, a more accurate study of how the prefetching hardware works, etc.

Another interesting issue is the study of the impact in area, access time and power consumption. This implies the implementation of the proposed cache organization using VLSI tools. This study could help to decide better ways of partitioning the cache or grouping the different submodules.

Finally, the last topic of interest is a hardware-based approach to set the hints in the LSMCache.

Software prefetching

An interesting topic to study is the interferences caused by spill code references. With the aim of reducing these interferences, a special buffer in which store the spill code data may be convinient. Some works have recently appear about a similar kind of buffer but handled as a complementary register file.

A main point to study in software prefetching is a mixed binding and non-binding prefetching scheme. The proposed algorithm (CSMS) is based on binding prefetching (or early scheduling), but as we have analyzed in Chapter 5, non-binding prefetching (or inserting prefetch instructions) can be beneficial in some cases (for instance, for memory instructions inside restrictive recurrences). An algorithm that uses the most convenient approach for each particular memory instruction should obtain better results.

Scheduling in clustered architectures

In this topic we are currently continuing the work developed in this thesis in some different lines. First, we have been improving the algorithm to schedule instructions in a semi-distributed clustered VLIW architecture (although could be adapted to the fully-distributed organization). The basic idea behind this improvement has been to include spill code. In the algorithm presented in Chapter 6, a cluster is not selected if no sufficient registers are available. With spill code, this restriction could be relaxed.

Another task will be the improvement of the algorithm by re-scheduling nodes (what is commonly called backtracking). This can help to obtain better schedules, although at the expense of increasing the scheduling time.

For the clustered VLIW architecture with distributed cache, in the organization proposed in Chapter 6 the cache was physically partitioned among the different clusters, and thus any address of the program in execution could be allocated in any cache. That required a hardware protocol, additional traffic (and more complexity in general) to keep the coherence among the different caches. We are studying other ways of partitioning the cache and algorithms to schedule instructions in such architectures.

REFERENCES

[1] S.G. Abraham, R.A. Sugumar, B.R. Rau and R. Gupta, “Predictability of Load/Store Instruction Latencies”, in *Procs. of 26th Int. Symp. on Microarchitecture (MICRO-26)*, pp.139-152, Dec. 1993

[2] V. Agarwal, M.S. Hrishikesh, S.W. Keckler and D. Burger, “Clock Rate versus IPC: The End of the Road For Conventional Microarchitectures”, in *Procs. of the 27th. Int. Symp. on Computer Architecture (ISCA’2000)*, pp. 248-259, June 2000

[3] G. Ammons, T. Ball and J.R. Larus, “Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling”, in *Procs. on of the 1997 Conf. on Programming Languages Design and Implementation (PLDI-97), 1997*

[4] E. Ayguadé, C. Barrado, A. González, J. Labarta, J. Llosa, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera and M. Valero, “Ictineo: a Tool for Research on ILP”, in *Supercomputing 96, Research Exhibit “Polaris at Work”*

[5] R. Bedichek, “Talismam: Fast and Accurate Multicomputer Simulation”, in *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp. 14-24, May 1995

[6] N. Bermudo, X. Vera, A. González and J. Llosa, “An Efficient Solver for Cache Miss Equations”, in *Procs. of Int. Symp. on Performance Analysis and System Software*, April 2000

[7] D. Bernstein, D. Cohen, A. Freund and D.E. Maydan, “Compiler Techniques for Data Prefetching on the PowerPC”, in *Procs. of Int. Conf. on Parallel Architectures and Compilation Techniques (PACT’95)*, pp.19-26, 1995

[8] D.C. Burger, J.R. Goodman and A. Kägi, “Memory Bandwidth Limitations of Future Microprocessors”, in *Procs. of 23th Int. Symp. on Computer Architecture (ISCA’96)*, May 1996

[9] D. Callahan, K. Kennedy and A. Porterfield, “Software Prefetching”, in *Procs. of the IV Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp.40.52, April 1991

[10] R. Canal, J.M. Parcerisa and A. González, “Dynamic Cluster Assigment Mechanisms”, in *Procs. of 6th Int. Symp. on High-Performance Computer Architecture*, pp. 133-142, Jan. 2000

[11] A. Capitanio, D. Dytt and A. Nicolau, “Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs”, in *Procs. of 25th. Int. Symp. on Microarchitecture (MICRO-??)*, pp. 192-300, 1992

[12] S. Carr, K.S. McKinley and C-W. Tseng, “Compiler Optimizations for Improving Data Locality”, in *Procs. of the V Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 252-262, Oct. 1994

[13] D. Culler and J.P. Singh, “Parallel Computer Architecture. A Hardware/Software Approach”, *Morgan Kaufmann Publishers, Inc.*, 1999

[14] Z. Cvetanovic and D. Bhandarkar, “Performance Characterization of the Alpha 21164 Microprocessor Using TP and SPEC Workloads”, in *Procs. of 2nd. Int. Symp. on High-Performance Computer Architecture (HPCA-2)*, pp. 270-280, 1996

[15] K.K. Chan, C.C. Hay, J.R. Keller, G.P. Kurpanek, F.X. Schumacher and J. Zheng, “Design of the HP PA 7200 CPU”, *Hewlett-Packard Journal*, Feb. 1996

[16] T-F. Chen and J-L. Baer, “Reducing memory latency via non-blocking and prefetching- caches”, in *Procs. V International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 51-61, Oct. 1992

[17] C-H. Chi and H. Dietz, “Unified Management of Registers and Cache Using Liveness and Cache Bypass”, in *Procs. of Int. Conf. on Programming Language Design and Implementation* (*PLDI’89)*, pp. 344-355, June 1989

[18] E. van der Deijl, G. Kanbier, O. Temam and E.D. Granston, “A Cache Visualization Tool”, *IEEE Computer*, 30(7), pp. 71-78, July 1997

[19] K. Diefendorff, R. Oehler and R. Hochsprung, “Evolution of the PowerPC Architecture”, in *IEEE Micro, Vol. 4, No. 2*, pp.34-49, 1994

[20] C. Ding, S. Carr, and P. Sweany, “Modulo scheduling with cache reuse information”, in *Procs. of EuroPar ’97*, pp. 1079-1083, August 1997

[21] J.H. Edmondson, P.I. Rubinfeld, P.J. Bannon, et al., “Internal organization of the Alpha 21164, a 300-Mhz 64-bit quad-issue CMOS RISC microprocessor”, in *Digital Technical Journal, Vol.7 No.1*, 1995

[22] A.E. Eichenberger, E.S. Davidson and S.G. Abraham, “Optimum Module Schedules for Minimum Register Requirements”, in *Procs. of Supercomputing 95*, pp.31-40, July 1995

[23] A.E. Eichenberger and E.S. Davidson, “Stage Scheduling: a Technique to Reduce the Register Requirements of a Module Schedule”, in *Procs. of 28th Int. Symp. on Microarchitecture (MICRO-28)*, pp.338-349, Nov.1995

[24] J. R. Ellis, “Bulldog: A Compiler for VLIW Architectures”, *MIT Press*, pp. 180-184, 1986

[25] P. Faraboschi, G. Brown, J. Fisher, G. Desoli and F. Homewood, “Lx: A Technology Platform for Customizable VLIW Embedded Processing”, in *Procs. of the 27th Int. Symp. on Computer Architecture (ISCA’2000)*, pp. 203-213, June 2000

[26] K.I. Farkas and N.P. Jouppi, “Complexity/performance tradeoffs with non-blocking loads”, in *Proc. 21th International Symposium on Computer Architecture (ISCA’94)*, pp. 211-222, 1994

[27] K.I. Farkas, P. Chow, N.P. Jouppi and Z. Vranesic, “The Multicluster Architecture: Reducing Cycle Time Through Partitioning”, in Procs. of *30th. Int. Symp. on Microarchitecture*, pp. 149-159, Dec. 1997

[28] M.M. Fernandes, J. Llosa and N. Topham, “Distributed Modulo Scheduling”, in *Procs. of Int. Symp. on High-Performance Computer Architecture*, pp. 130-134, Jan. 1999

[29] M. Franklin, “The Multiscalar Architecture”, *PhD Thesis, Technical Report TR-1196, Computer Science Dept., UW-Madison*, 1993

[30] J. Fridman and Zvi Greefield, “The TigerSharc DSP Architecture”, *IEEE Micro*, pp. 66-76, Jan-Feb. 2000

[31] D. Gannon, W. Jalby and K. Gallivan, “Strategies for Cache and Local Memory Management by Global Program Transformations”, *Journal of Parallel and Distributed Computing, 5*, pp. 587-616, 1988

[32] J. Gee, M. Hill, D. Pnevmatikatos and A.J. Smith, “Cache Performance of the SPEC92 Benchmark Suite”, *IEEE Micro*, pp. 17-27, Aug. 1993

[33] S. Ghosh, M. Martonosi and S. Malik, “Cache Miss Equations: an Analytical Representation of Cache Misses”, in *Procs. of Int. Conf. on Supercomputing (ICS’97)*, pp. 317-324, July 1997

[34] A.J. Goldberg and J. Hennessy, “Performance Debugging Shared Memory Multiprocessor Programs with Mtool”, in *Procs. of Supercomputing’91 Conf. (SC’91)*, pp. 481-490, 1991

[35] S. Goldschmidt and J. Hennessy, “The Accuracy of Trace-Driven Simulation of Multiprocessors”, in *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp 146-157, May 1993

[36] A. González, C. Aliagas and M. Valero, “A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality”, in *Proc. of Int. Conf. on Supercomputing (ICS’95)*, pp. 338-347, 1995

[37] E.H. Gornish, E.D. Granston and A.V. Veidenbaum, “Compiled-Directed Data Prefetching in Multiprocessors with Memory Hierarchy”, in *Procs. of 17th Int. Symp. on Computer Architecture (ISCA’90)*, pp.354-368, 1990

[38] R. Govindarajan, E.R. Altman and G.R. Gao, “Minimal Register Requirements Under Resource-Constrained Software Pipelining”, in *Procs. of 27th Int. Symp. on Microarchitecture (MICRO-27)*, pp.85-94, Nov. 1994

[39] L. Gwennap, “Digital 21264 Sets New Standard”, *Microprocessor Report, 10(14)*, Oct. 1996

[40] L. Gwennep, “Alpha 21364 to Ease Memory Bottleneck”, *Microprocessor Report, 12(14)*, pp. 12-15, Oct. 1998

[41] S. Hadjiyannis, M. Tomasko and W. Najjar, “An Evaluation of Split Scalar/Array Caches”, *Technical Report CS-TR-97-104, CS Department, Colorado State University*, Jan. 1997

[42] D.T. Harper III and D.A. Linebarger, “A Dynamic Storage Scheme for Conflict Free Vector Access”, *Procs. of the 14th. Int. Symp. on Computer Architecture (ISCA’87)*, pp. 72-77, 1987

[43] J.L. Hennessy and D.A. Patterson, “Computer Architecture. A Quantitative Approach”, *Morgan Kaufmann Publishers, 2nd. Edition, San Francisco*, 1996

[44] M.D. Hill, “Aspects of Cache Memory and Instruction Buffer Performance”, *PhD. Thesis, University of California at Berkeley, UCB/CSD 87/381*, Nov. 1987

[45] A.S. Huang and J.P. Shen, “A Limit Study of Local Memory Requirements Using Value Reuse Profiles”, in *Procs. of 28th Int. Symp. on Microarchitecture (MICRO-28)*, pp. 71-81, 1995

[46] R.A. Huff, “Lifetime-Sensitive Modulo Scheduling”in *Procs. of Int. Conf. on Programming Language Design and Implementation (PLDI’93)*, pp.318-328, 1993

[47] D. Hunt, “Advanced performace features of the 64-bit PA-8000”, in *Compcon Digest of Papers*, pp. 123-128, 1995

[48] Intel Corp., “IA-64 Application Developer’s Architecture Guide”, *Intel Corporation Report*, May 1999

[49] S. Jain, “Circular Scheduling: a New Technique to Perform Software Pipelining”, in *Procs. of Int. Conf. on Programming Language Design and Implementation (PLDI’91)*, pp.219-228, June 1991

[50] S. Jang, S. Carr, P. Sweany and D. Kuras, “A Code Generation Framework for VLIW Architectures with Partitioned Register Banks”, in *Procs. of 3rd. Int. Conf. on Massively Parallel Computing Systems*, April 1998

[51] T. Johnson and W.W. Hwu, “Run-Time Adaptive Cache Hierarchy Management via Reference Analysis”, in *Procs. of 24th Int. Symp. on Computer Architecture (ISCA’97)*, pp. 315-326, June 1997

[52] N.P. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”, in *Procs. of 17th Int. Symp. on Computer Architecture (ISCA’90)*, pp. 364-373, 1990

[53] T. Juan, J.J. Navarro and T. Lang, “Removing Interferences Misses Using Cache Bypass Buffers”, *Technical Report UPC-CEPBA-94-14*, 1994

[54] K. Kailas, K. Ebcioglu and A. Agrawala, “CARS: A New Code Generation Framework for CLustered ILP Processors”, in *Procs. 7th Int. Symp. on High-Performance Computer Architecture (HPCS-7)*, Jan. 2001

[55] V. Kathail, M. Schlansker and B. Rau, “HPLabs PlayDoh Architecture Specification: Version 1.0”, *Technical Report HPL-93-80, Hewlett-Packard Labs*., March 1994

[56] G.A. Kemp and M. Franklin, “PEWs: A Decentralized Dynamic Scheduler for ILP Processing”, in *Procs. on Int. Conf. on Parallel Processing*, pp. 239-246, Aug. 1996

[57] K. Kennedy, D. Callahan and A. Porterfield, “Analyzing and Visualizing Performance of Memory Hierarchy”, in *Instrumentation for Visualization*, ACM Press, New York, 1990

[58] D.R. Kerns and S.J. Eggers, “Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain”, in *Procs. of Int. Conf. on Programming Language Design and Implementation (PLDI’93)*, pp.278-289, 1993

[59] A.C. Klaiber and H.M. Levy, “An Architecture for Software-Controlled Data Prefetching”, in *Procs. of 18th Int. Symp. on Computer Architecture (ISCA’91)*, pp.43-53, May 1991

[60] D. Kroft," Lockup-Free Instruction Fetch/Prefetch Cache Organization", in *Procs. 8th Int. Symp. on Computer Architecture*, pp. 81-87, 1981

[61] M.S. Lam, "Software Pipelining: an Effective Scheduling Technique for VLIW Machines", in *Procs. of Int. Conf. on Programming Language Design and Implementation (PLDI'88)*, pp.318-328, June 1988

[62] D.M. Lavery and W.W. Hwu, "Unrolling-Based Optimizations for Modulo Scheduling", in *Procs. of 28th. Int. Symp. on Microarchitecture*, pp., 1995

[63] A.R. Lebeck and D.A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study", *IEEE Computer,* 27(10), pp. 15-26, Oct. 1994

[64] A.R. Lebeck and D.A. Wood, "Fast-Cache: a New Abstraction for Memory-System Simulation" in *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp. 220-230, 1995

[65] J. Llosa, M. Valero, E. Ayguadé and A. González, "Hypernode Reduction Modulo Scheduling", in *Procs. of 28th Int. Symp. on Microarchitecture (MICRO-28*), pp.350-360, Nov. 1995

[66] J. Llosa, A. González, E. Ayguadé and M. Valero, "Swing Modulo Scheduling", in *Procs. of Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'96)*, pp.80-86, Oct. 1996

[67] P. Magnusson, "A Design for Efficient Simulation of a Multiprocessor", in *Procs. of the Western Simulation Multiconference on Int. Workshop on MASCOTS-93*, pp. 69-78, La Jolla, California, 1993

[68] "MAP1000 unfolds at Equator", *Microprocessor Report, 12(16)*, Dec. 1998

[69] P. Marcuello and A. González, "Clustered Speculative Multithreaded Processors", in *Procs. on the 13th Int. Conference on Supercomputing (ICS'99)*, pp. 365-372, June 1999

[70] M. Martonosi, A. Gupta and T. Anderson, "Memspy: Analyzing Memory Performance System Bottlenecks in Programs", *Performance Evaluation Rev.*, 20(2), June 1992

[71] D. Matzke, "Will Physical Scalability Sabotage Performance Gains", *IEEE Computer, Vol. 30, No. 9*, pp. 37-39, Sept. 1997

[72] K. McKinley and O. Temam, "A Quantitative Analysis of Loop Nest Locality", in *Procs. of VII Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 94-104, Oct. 1996

[73] V. Milutinovic, B. Markovic, M. Tomasevic and M. Tremblay, "The Split Temporal/Spatial Cache: Initial Performance Analysis", in *Procs. of SCIzzL-5 Workshop*, pp. 63-70, March 1996

[74] MIPS, "RISCompiler Languages Programmer's Guide", *MIPS*, 1988

[75] MIPS, "R10000 Microprocessor User's Manual", *MIPS Technologies, Inc.*, June 1995

[76] T.C. Mowry, M.S. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", in *Procs. of the V Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp.62-73, Oct. 1992

[77] E. Nystrom and A. E. Eichenberger, "Effective Cluster Assingment for Modulo Scheduling", in *Procs. of 31th. Int. Symp. on Microarchitecture*, pp.103-114, 1998

[78] E. Özer, S. Banerjia and T.M. Conte, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures", in *Procs. of 31st Int. Symp. on Microarchitecture*, pp. 308-315, Nov. 1998

[79] S. Palacharla and R.E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement", in *Procs. of the 21st Int. Symp. on Computer Architecture (ISCA-21)*, pp. 24-33, Apr. 1994

[80] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors", in *Procs. of the 24th. Int. Symp. on Computer Architecture (ISCA'97)*, pp. 1-13, June 1997

[81] D. B. Papworth, "Tuning the Pentium Pro Microarchitecture", *IEEE Micro, 16(2)*, pp. 8-15, April 1996

[82] B.R. Rau and C.D. Glaeser, “Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing”, in *Procs. on the 14th Ann. Workshop on Microprogramming*, pp. 183-198, Oct. 1981

[83] B.R. Rau, “Iterative Modulo Scheduling: an Algorithm for Software Pipelining Loops”, in *Procs. of 27th Int. Symp. on Microarchitecture (MICRO-27)*, pp.63-74, Nov. 1994

[84] S. Reinhardt, M. Hill, J. Larus, A. Leveck, J. Lewis and D. Wood, “The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers”, in *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp. 48-60, May 1993

[85] G. Rivera and C-W. Tseng, “Data Transformations for Eliminating Conflict Misses” in *Procs. of Conf. on Programming Language Design and Implementation (PLDI‘98)*, 1998

[86] J.A. Rivers and E.S. Davidson, “Reducing Conflicts in Direct-Mapped Caches with Temporality-Based Design”, in *Procs. of Int. Conf. on Parallel Processing (ICPP’96)*, pp. 93-103, Dec. 1996

[87] J.A. Rivers, S. Tam, G.S. Tyson and E.S. Davidson, “Utilizing Reuse Information in Data Cache Management”, in *Procs. of Int. Conf. on Supercomputing (ICS’98), July 1998*

[88] E. Rotenberg, Q. Jacobson, Y. Sazeides and J.E. Smith, “Trace Processors”, in *Procs. of the 30th Int. Symp. on Microarchitecture (MICRO-30)*, pp. 138-148, Dec. 1997

[89] J. Sánchez, A. González and M. Valero, “Static Locality Analysis for Cache Management”, in *Procs. of Int. Conf. on Parallel Architectures and Compilation Techniques (PACT’97)*, Nov. 1997

[90] J. Sánchez and A. González, “Cache Sensitive Modulo Scheduling”, in *Procs. of 30th. Int. Symp. on Microarchitecture*, pp. 338-348, Dec. 1997

[91] J. Sánchez and A. González, “The Effectiveness of Loop Unrolling for Modulo Scheduling in Clustered VLIW Architectures”, to appear in *Procs. of the 29th. Int. Conf. on Parallel Processing*, Aug. 2000

[92] S.S. Sastry, S. Palacharla and J.E. Smith, “Exploting Idle Floating-Point Resources for Integer Execution”, in *Procs. of Int. Conf. on Programming Languages Design and Implementation*, pp. 118-129, June 1998

[93] Semiconductor Industry Association, “The National Technology Roadmap for Semiconductors: Technology Needs”, 1997

[94] A.J.. Smith, “Cache Memories”, *Computing Surveys, Vol. 14, No. 3*, pp. 473-530, Sept. 1982

[95] G. Sohi, S.E. Breah and T.N. Vijaykumar, “Multiscalar Processors”, in *Procs. of the 22nd. Int. Symp. on Computer Architecture (ISCA’95)*, pp.414-425, June 1995

[96] Standard Performance Evaluation Corporation, “The SPEC95 benchmark suite”, *http://www.specbench.org*, 1995

[97] J.M. Stone and R.P. Fitzgerald, “Storage in the PowerPC”, *IEEE Micro, vol. 15, no. 2*, pp. 50-58, April 1995

[98] R. Sugumar, “Multi-configuration simulation algorithms for the evaluation of computer designs”, *PhD dissertation, University of Michigan*, 1993

[99] O. Temam, E.D. Granston, W. Jalby, “To Copy or not to Copy: A Compile-time Technique for Assessing when Data Copying Should be Used to Eliminate Cache Conflicts”, in *Procs. of Supercomputing’93 Conf. (SC’93)*, pp. 410-419, 1993

[100] O. Temam, C. Fricker and W. Jalby, “Cache Interference Phenomena”, in *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, May 1994

[101] Texas Instruments Inc., “TMS320C62x/67x CPU and Instruction Set Reference Guide”, 1998

[102] Texas Instruments Inc., “TMS320C6211 Cache Analysis”, *Application Report SPRA472*, Sept. 1998

[103] M. Tremblay and J.M. O’Connor, “UltraSparc I: a Four-Issue Processor Supporting Multimedia”, *IEEE Micro, 16(2)*, pp. 42-49, April 1996

[104] G. Tyson, M. Farrens, J. Matthews and A. Pleszkun, “A Modified Approach to Data Cache Management”, in *Proc. of 28th Int. Symp. on Microarchitecture (MICRO-28)*, pp. 93-103, Dec. 1995

[105] R.A. Uhlig and T.N. Mudge, “Trace-driven Memory Simulation: a Survey”, *ACM Computing Surveys*, 1997

[106] S. Vajapeyam and T. Mitra, “Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences”, in *Procs. of Int. Symp. on Computer Architecture (ISCA’97)*, pp. 1-12, June 1997

[107] X. Vera, J. Llosa, A. González and C. Ciuraneta, “A Fast Implementation of Cache Miss Equations”, in *Procs. of the 8th. Int. Workshop on Compilers for Parallel Computers*, pp. 319-326, Jan. 2000

[108] J. Wang and C. Eisenbeis, “Decomposed Software Pipelining: a New Approach to Exploit Instruction Level Parallelism for Loops Programs”, in *IFIP*, Jan. 1993

[109] M. Wilkes, “Slave Memories and Dynamic Memory Allocation”, *IEEE Transaction on Electronic Computers, EC-14(2)*, pp. 270-271, April 1965

[110] E. Witchel and M. Rosenblum, “Embra: Fast and Flexible Machine Simulation”, in *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, May 1996

[111] M.E. Wolf and M.S. Lam, “A Data Locality Optimizing Algorithm”, in *Procs. of Int. Conf. on Programming Language Design and Implementation (PLDI’91)*, pp.30-44, 1991

[112] K.C. Yeager, “The MIPS R10000 Superscalar Microprocessor”, *IEEE Micro, 16(2)*, pp. 28-40, April 1996

[113] V.V. Zyuban, “Low-Power High-Performance Superscalar Architectures”, *PhD Thesis, Dept. of Computer Science and Engineering, University of Notre Dame*, Jan. 2000