ADVANCED VECTOR ARCHITECTURES

ADVANCED VECTOR ARCHITECTURES

Roger Espasa

UPC. Universitat Politècnica de Catalunya

Barcelona (Spain). February, 1997

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

ADVANCED VECTOR ARCHITECTURES

Roger Espasa

Approved, Thesis Committe:

Jesús Labarta ....

Emilio López Zapata ....

Emilio Luque ....

Juan José Navarro ....

James E. Smith ....

Als meus pares, Ramon i Carme,
a la meva germana Marina i,
molt especialment, a la Marta.

ABSTRACT

Vector architectures have long been the architecture of choice for numerical high performance computing. Their large memory bandwidth and the ability to tolerate relatively long memory latencies have been central to their success in executing numerical applications with very large data sets.

Recently, though, vector processors have been slowly displaced by other supercomputing paradigms based on large or medium scale multiprocessing that use as a building block commodity superscalar CMOS microprocessors. Vector machines have not improved fast enough to keep a good price/performance ratio and have been relegated to those types of problems requiring very large bandwidths and/or very large datasets. Moreover, the performance achieved by real programs on vector machines has usually been far from peak speed. This has inclined users to believe that the extreme cost of vector machines is not justified in front of more affordable microprocessor based machines.

The difficulty in bringing vector supercomputers back to a competitive price/performance point is in their memory systems, which, although provide the uttermost performance, are extremely expensive. The use of many interleaved parallel banks and static memory storage leads to very high bandwidths at a reasonable latency but incurs in large system costs. In order to reduce the cost of vector supercomputers, the memory system should be made more economical by moving from static to dynamic storage technology. Besides, vector machines should also provide increased performance to clearly distinguish themselves from commodity superscalar micros.

These two goals are somewhat in contradiction since moving from static to dynamic storage could double or triple memory latency due to the intrinsically larger response time of dynamic memory. Although the vector paradigm does provide some latency

tolerance, we will start by showing that it is not enough for truly large latencies and that a vector processor would suffer a severe performance degradation under such conditions.

This thesis will show that both the lack of tolerance to very large memory latencies and the discrepancy between actual and peak performance of vector machines originate in the conservative in-order instruction execution model currently in use. This in-order model results in a large underutilization of the high memory bandwidth provided by the memory system. The in-order model is also responsible for not requesting data early enough in time to mask large memory latencies.

Three solutions to the latency and performance problems will be presented. Using dynamic scheduling techniques already exploited in superscalar machines, we will show that performance of traditional vector architectures can be greatly improved while, at the same time, providing the necessary latency tolerance to compensate for slow memory systems. The first two techniques, *decoupling* and *out-of-order execution*, look for independent operations of a singe program stream to be executed in parallel. The third technique, *multithreading*, will improve global throughput by interleaving independent instructions from different programs.

Alone or combined, these three techniques can yield better performance and allow the use of cheaper main memory systems which can make vector architectures regain some of their lost prominence.

Spill code is an important problem in architectures with a limited number of registers. Vector machines have traditionally been built using only a few number of registers and, therefore, experience this spill problem. This thesis will also study the amount of spill code present in vector programs and will present three different techniques that dynamically eliminate spill loads and stores, thereby reducing total memory traffic and improving execution time.

Our goal is to present solutions, both to the latency and spill problems, that are fully binary compatible with existing vector instruction sets. Thus, the techniques described

in this thesis are fully transparent to the end user and do not require recompilation nor the addition of special opcodes to the instruction set architecture.

ACKNOWLEDGEMENTS

I would like to thank Mateo Valero, my thesis advisor, for his guidance of this dissertation and his support. I would also like to thank James E. Smith for many fruitful discussions and for encouraging the out-of-order work presented here. I am also indebted to him for hosting me during my stay in Madison, and for many good times together, including some excellent paella demonstrations. My gratitude also to Eduard Ayguadé and David Padua, who helped in the initial steps of this thesis.

I am especially grateful to Toni Juan for many insightful discussions, for his patience in reading my boring manuscripts and for his encouragement in so many different situations. I also would like to thank Montse Peiron for all her support and help, for giving me some very good advice (which I didn't follow, by the way) and for constantly encouraging me to finish this work.

Toni Cortés has been an excellent office-mate along this four years and I thank him for help and feedback on so many different topics.

I would also like to thank all my colleagues in the Computer Architecture Department for their support and for putting up with all my time-consuming simulations that have eaten more than 65 million cpu seconds in the last four years. Also thanks to the excellent technical support provided by Rosa Castro from the LCAC and Oriol Riu and Judit Jiménez from the CEPBA.

During my academic life I have encountered many teachers among whom I would like to remember Pere Ribera, who has given me much more than I could ever express in a few words. My best regards also for Diana Garrigosa and Manel Queralt, who introduced me to the world of computers and who are probably to blame for my early interest in these strange machines. Last, but not least, I have to thank Pere Cairó for he gave me the pleasure of science and research, and was the first to teach me that discovering new things, no matter how insignificant, can be a very exciting activity.

Special thanks go to Pau Calpe and Gonzalo Martín for being truly good friends all these years, for helping in difficult situations and for all the good times we have had together. Ignasi Biosca, Lluis Domènech, David Font and Joan Gispert have also been great friends, great basketball players and better Falstaff companions.

Finally, I would like to thank my parents, Ramon and Carme, my sister Marina, my grandmother Carme and all my family for their love and support along these years. Their dedication and encouragement has made possible this work. My deepest thanks to Marta, for more than I can say. This work is dedicated to them and to the memory of my grandmother Elisa.

This work was supported in part by the Ministry of Education of Spain under contracts TIC 880/92 and TIC 0429/95, by CIRIT grant BEAI96/II/124 and by the CEPBA (European Center for Parallelism of Barcelona).

CONTENTS

1

INTRODUCTION

Summary

In this chapter we present the motivations behind this thesis, we make a case for studying vector architectures despite the relative decline of their success and we discuss the latency problem both in the context of vector and superscalar architectures. We outline the three solutions to the latency problem studied in this thesis: decoupling, out-of-order execution and multithreading. We also introduce the spill code problem found in vector machines with limited number of registers.

1.1 MOTIVATION

Vector architectures have traditionally been the architecture of choice for building supercomputers. The first vector machines were supercomputers using memory-to-memory operation [HT72, Wat72], but vector machines only became commercially successful with the addition of vector registers in the Cray-1 [Rus78]. Following the Cray-1, a number of vector machines have been designed and sold, from supercomputers with very high vector bandwidths [Oed92, KIS+94] to more modest mini-supercomputers [Con92, PM86].

The peak performance of vector supercomputers has been constantly improving from the original 160 Mflops per processor of the Cray-1 up to 5.5 Gflops per processor of the more recent NEC SX-3 [IW91]. This improvement has been achieved both through better cycle times (from 12.5ns in the Cray-1 down to 2.9ns in the SX-3) and higher number of floating point operations initiated per cycle (from 2 flops/cycle in the Cray-1 up to 4 in the Cray C-90 [Oed92] and 16 in the SX-3).

Recently though, parallel vector processors (PVP) have been slowly displaced by other forms of supercomputing. The most notable ones have been large (MPP) and medium (SMP) scale multiprocessors built out of commodity microprocessors. The success of these other technologies has been mostly based on leveraging CMOS microprocessor technology and DRAM main memory systems, yielding systems that have a high performance at a low cost. This has led to a shrinkage of the traditional PVP market niche. Only applications that (1) can be highly vectorized and (2) have very large data sets and/or require large amounts of bandwidth are currently targeted to PVP's.

Why have vector architectures been slowly outshaded by other forms of computing ? Four reasons can be pointed out:

First, because of the high cost of traditional vector processors. In order to keep the powerful cpu fed with enough data two related problems have to be solved. It is necessary to provide enough bandwidth from the memory system to the processor to ensure that the maximum processing rate is achieved. Moreover, this bandwidth has to be provided without increasing too much the latency delays that each memory request has to pay. Vector processors rely on high-performance, highly-interleaved memory systems that can sustain a bandwidth rate in accordance with their computing capabilities. Through the use of many parallel banks of interleaved memory (between 256 and 1024 banks depending on processor and configuration [SWL+92]) the necessary bandwidth is usually achieved. To keep the latency reasonably low, in the range of 10–30 processor cycles, the fastest memory technology must be used. Vector processors tend to have SRAM/SSRAM memory modules [Pri96] with cycle times in the 10–20ns range. This memory architecture has a very high cost and, usually, turns out to be a very large fraction of the overall system cost. In comparison, other supercomputing paradigms, such as SMP, leverage commodity technology: they use cache-based microprocessors and bus-based DRAM main memory systems. This yields a very low cost and delivers a very good price/performance ratio. Larger MPP systems might require special purpose interconnects but still use as a building block cheap CMOS microprocessors and DRAM main memories.

Second, vector supercomputers have been suffering from the so-called "attack of the killer micros". Scalar processors using CMOS technology have been evolving very rapidly and their speed has been increasing at a rate of 60% per year [PAY96]. These scalar processor have included a variety of architecture techniques, such as decoupling [Smi82b], out-of-order execution [Tom67, Yag96] and hierarchical memory systems based on caches [KELS62, Wil65, Hil87, Hil88, Smi82a, SG83, HP90, LRW91a], that have greatly improved their processing power. Moreover, the internal instruction level parallelism of these processors has also increased, resulting in microprocessor performance that has approached that of supercomputers at a much lower cost. For example, the recent Alpha 21264 is rated somewhere over 1 Gflop [Kel96] and its cost will probably be ten times lower than the cost of a single SX-3 processor. This has led to a situation where, for certain classes of cache-friendly problems, a microprocessor can match and sometimes exceed the performance of a vector supercomputer at a small fraction of its cost. While it is true that not all problems are well suited for cache-based machines, it is clear that these types of systems have taken a substantial share of applications that had previously belonged to the vector computing realm. As a result, vector processors have been relegated to those classes of problems where the data set size is so big that it will not fit in a cache-based memory system.

Third, the slow decline of vector processors can also be attributed to the fact that their architecture has not changed much since the introduction of the Cray-1 [Rus78]. While superscalar microprocessors have adopted many architectural features to increase performance while still retaining low cost, vector machines have used almost the same architectural concepts introduced in 1976. In order to remain viable, and to be extended to low cost systems, vector designs must somehow evolve.

Fourth, despite the theoretical processing rates of vector cpus and the high bandwidth memory systems built around them, a constant in the evolution of vector supercomputers has been that the achieved performance of real programs has always been far from the theoretical peak performance [SH91, Don93, SH94, Del91, PB88]. What is worse is that this discrepancy between peak and actual performance occurs even for programs that are highly vectorized. As a result, the theoretical advantage of the vector computing paradigm over other forms of supercomputing is lost.

Can vector architectures evolve to overcome these problems or are they condemned to disappear ? We see three paths that vector machines can follow to regain some popularity:

- Deliver sustained performance truly close to peak performance (thereby justifying a high cost), or
- Deliver the current (not so good) performance but significantly reduce total system cost, or
- Both deliver sustained performance close to peak performance *and* reduce total system cost.

The proposals in this thesis are aimed at the third path. We will start by analyzing a traditional vector architecture and we will show that the reason it falls short of expected performance is the poor usage of its expensive memory system. Then we will present three different proposals to overcome this problem. These solutions will, as a byproduct, provide a way to reduce overall system cost by reducing the cost of the most expensive part of a vector machine: its memory system.

1.2 WHY STUDY VECTOR ARCHITECTURES ?

Given all the reasons mentioned above for the slow decline of vector architectures one could legitimately ask why should we study vector architectures any more. The reason is that vector instruction sets do have some inherent advantages over scalar instruction sets. Although these advantages might not be clear enough in current *implementations* of vector architectures, we believe that the evolution of current integrated circuit technology will only contribute to make these advantages more clear.

For example, vectors have inherent advantages when it comes to memory usage. A single instruction can exactly specify a long sequence of memory addresses. These may be consecutive, may be separated by some fixed stride length, or may be irregularly placed in memory. Consequently, the hardware has considerable advance knowledge regarding memory references, can schedule these accesses in an efficient way, and needs to access no more data than is actually needed. In addition, a vector memory operation is able to amortize startup latencies over a potentially long stream of vector elements. These features would seem to make vector architectures ideal for the current trend toward longer memory latencies (measured in processor cycles).

Also, the relatively low instruction fetch bandwidth required by vector processors can also be an advantage in today's technology environment. With off-chip references costing many cycles and superscalar performance being limited by instruction fetch and decode bandwidth [BGK96], the ability to specify in a single instruction a large number of operations can be very beneficial.

Moreover, the value of vector architectures for desktop applications is recently being recognized. In particular, many DSP and multimedia applications – graphics, compression, encryption – are very well suited for vector implementation [NMP+96, ABI+95]. In a multimedia application, vector instructions are implemented in a multimedia co-processor with access via library calls. Or, alternatively, short vectors are held in memory of a general purpose processor, and are operated on by multimedia extension instructions [Lee96, PW96]. These short vectors lack some of the flexibility of traditional vector architectures, but this is likely to change as multimedia instruction sets evolve [Han96, TONH96]. Also, research focusing on new processor-memory organizations, such as IRAM [PAY96], would also benefit from vector technology.

All in all, we believe that, far from being dead, vector architecture will reappear in the short term in several fields. They will surely evolve from the ECL mainframe-based style of machines into small, CMOS coprocessors that will help a general purpose microprocessor perform some of its compute-intensive tasks.

1.3 THE PERFORMANCE GAP

What are the reasons for the performance gap between actual and peak performance in a vector processor ? A certain number of factors have been traditionally pointed out in previous studies [Del91, BCG+91, SWL+92, Fat89, SHH90]. The degree of vectorization of programs is a key parameter that determines the maximum speedup achievable through vector processing. Amdahl's law [Amd67] shows that a balanced vector and scalar performance is also necessary to achieve high performance. Execution startup penalties and functional unit latencies are also factors that may limit performance.

Nonetheless, one of the major factors affecting performance has not been thoroughly researched and will be the main motivation of this thesis: *traditional vector architectures are not able to take full advantage of the large, flat memory bandwidths they are provided with.*

Several studies in recent years [SH94, EVP+95] have stressed this particular point. Functional unit hazards and conflicts in the vector register file ports can make vector processors stall for long periods of time and result in latency problems similar to those in scalar processors. In [EVP+95] it is shown that the memory port of a single-port vector computer was heavily underutilized even for programs that were memory bound. It also shows how a vector processor could spend up to 50% of all its execution cycles waiting for data to come from memory. In a similar line, work by [RR94] shows how the memory port usage of Cray Y-MP system is also very low, sustaining an average of 1.0 memory transactions per cycle (where the machine could sustain 3 transactions per cycle on its two load and one store memory ports).

This bandwidth underutilization translates in two cascaded effects:

- First, since the available bandwidth is not fully used, vector architectures are not very latency tolerant. Increases in main memory latency result in a degradation in execution time.
- As a consequence, in order to keep their expected performance, vector architectures are tied to high performance, high cost SRAM memory modules.

Contrary to what we are suggesting, vector machines have traditionally been considered to have latency tolerance properties. Since each vector instruction works on a long stream of operations, functional unit latencies and memory latencies can potentially be amortized across all vector elements. In the particular case of memory accesses, once a memory load operation is started, it pays for some initial (potentially long) latency, but then, assuming no bank conflicts, it can deliver one word per machine cycle.

While this latency amortization is undoubtedly true, we will present data showing that in the face of large memory latencies, vector processors can suffer from severe performance degradation. We will show that this is due to the simplicity in instruction dispatching of traditional vector architectures. Each hazard occurring in the cpu, such as a functional unit conflict, causes a stall in the instruction dispatcher. If the stall occurs *while the memory ports are idle*, the available memory bandwidth goes unused. Furthermore, latency tolerance properties of vectors are lost: the next following load instruction at an idle memory port will expose the full memory latency.

If vector processors could be made more latency tolerant, then their memory systems could be implemented in cheap CMOS technology, thereby reducing total system cost. The gains in the price reduction of the memory system would certainly offset the extra hardware inside the CPU needed to achieve this latency tolerance.

1.4 SUPERSCALAR ARCHITECTURES AND THE LATENCY PROBLEM

Superscalar architectures do also suffer from the effects of the latency problem. In that context, the disparity between the CPU speed and main memory speed has been even larger than in the vector supercomputer world. Technological improvements in cpu speed have not been matched by similar improvements in memory chips. Figure 1.1 shows the evolution, in relative terms, of microprocessors and DRAM speeds over the past 16 years. As it can be seen, from 1986 onward the microprocessor cycle time has been improving at a rate of 60% per year, approximately. Meanwhile, DRAM speeds have been slowly improving at a rate of only 7% per year. Producers of DRAM's have mostly focused on improving cost per bit and density per chip rather than favoring faster cycle times. The result is that, although density has gone from 16 Kbits in a chip in 1976 to 64 Mbits in a chip in 1996 (a 4000-fold improvement), these same chips have only moved from a 400 ns cycle time down to a 100 ns cycle time (a 4-fold improvement) [Pri96, page 96].

Also, the instruction level parallelism available in recent microprocessors has increased. Since several instructions are being issued at the same processor cycle, the total amount of data requested per cycle to the memory system is much higher.

These two factors have led to a situation where memory chips are on the order of 10 to a 100 times slower than cpus and where each off-chip memory access can have extremely large

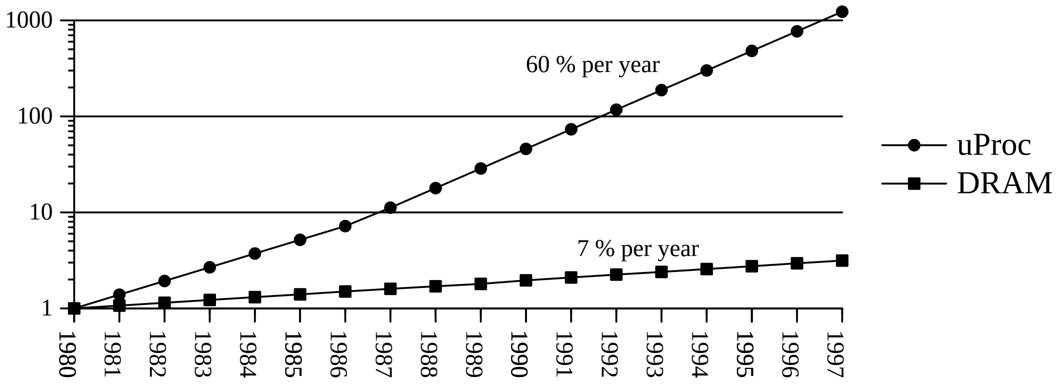

Figure 1.1 Relative evolution of microprocessor and DRAM speeds (source [PAY96]).

latencies. As a result, and depending on the degree of cache-friendliness, the total execution time of a program can be greatly dominated by average memory access time.

Current superscalar processors have been attacking the memory latency problem through basically four main types of techniques: caching, decoupling, out-of-order execution and multithreading (these, sometimes, may appear together).

Caching

Cache-based superscalar processors reduce the average memory access time by placing the working set of a program in a faster level in the memory hierarchy [KELS62, Wil65, Smi82a, SG83, Hil87, Hil88]. A thorough research on cache organization and parameters has led to very low miss rates in current commercial microprocessors [BW88, PHH88, WBL89, BKB90, AP93, Sez93, BCcRJ$^+$94, PK94, CG95, WSY95]. See [EP88, Jou93] for a discussion on internal cache parameters, such as the interaction of write policy and line size. Software and hardware techniques such as [Goo83, CB92, CB94a, MLG92] have been devised to *prefetch* data from lower levels in the memory hierarchy to higher levels (closer to the cpu) before the data is actually needed. On top of that, program transformations such as loop blocking [PHH89, WL91, KM92, LRW91b, Li95, CM95] have proven very useful to fit the working set of a program into multilevel memory hierarchies.

Related to data caching, software pipelining [Lam88, GHW90, GAG94, Jai91, RLTS92, Ram94, Rau94] has also contributed to hide memory latency and the penalties associated with cache misses by overlapping several iterations of a single loop.

Decoupling

Decoupled scalar processors [SWP86, Smi84, KHC94] have focused on numerical computation and attack the memory latency problem by making the observation that the execution of a program can be split into two different tasks: moving data in and out of the processor and executing all arithmetic instructions that perform the program computations. A decoupled processor typically has two independent processors (the address processor and the computation processor) that perform these two tasks asynchronously and that communicate through architectural queues. Latency is hidden by the fact that usually the address processor is able to slip ahead of the computation processor and start loading data that will be needed soon by the computation processor. This excess data produced by the address processor is stored in queues, and stays there until it is retrieved by the computation processor.

Out-of-Order Execution

Dynamic instruction issue is used in scalar processors to attack the memory latency problem by allowing memory reference instructions to proceed when other instructions are waiting for memory data. That is, memory reference instructions are allowed to slip ahead of execution instructions. This characteristic is made very explicit in decoupled scalar processors [SWP86, Smi84, KHC94], but is present in all the dynamically scheduled scalar processors.

Out-of-order execution was already present in machines such as the IBM 360/91 [AST67], which had a set of intermediate buffers that decoupled the core cpu from the memory access and used a renaming mechanism by Tomasulo [Tom67] to allow operations execute fully out-of-order. Current out-of-order micros [Yag96, TGN95, CS95, BDHS94, Kel96] use several variations of the original renaming algorithm to achieve the same effect.

Multithreading

Multithreaded scalar processors [Aga92, TEE$^+$95, TEL96, HKN$^+$92, EJK$^+$96] attack the memory latency problem by switching between threads of computations every time a long-latency operation (such as a cache miss) threatens to halt the processor. This approach not only fights memory latency, but also produces a system with higher throughput and better resource utilization. The change of thread on long latency operations implies an increase in exploitable parallelism, a decrease in the probability of halting the cpu due to a hazard, and yields a higher occupation of the functional units and an improved system throughput. While each single thread still pays latency delays, the cpu is (presumably) never idle thanks to this mixing of different threads of computation.

1.5 VECTOR ARCHITECTURES AND THE LATENCY PROBLEM

Can vector architectures borrow some of the techniques used in superscalar microprocessors to overcome the latency problem ?

Let's look at techniques in widespread use in current superscalar microprocessors. Caches and software pipelining are two techniques that have been studied [KSF$^+$94, Lam88, TDT88, MSAD91a] in the context of vector processors but that have not been proved useful enough to be in widespread use in current vector machines. Caches have traditionally not been used in vector architectures because working sets of vector programs are very large, and vector loads tend to produce "sweeps" over the cache that make all its data useless [KSF$^+$94].

Software pipelining [Lam88] has the effect of enlarging register lifetimes, which translates in a higher number of registers needed to hold live variables and thus produces a high pressure on the register allocator. Although in [TDT88, EJL88, MSAD91a] software pipelining is applied to vector computers, it is difficult to use it in a general context if the vector register file has a limited number of registers.

Data prefetching [FP91b] has also not been used in vector processors due to the high cost of an incorrect prefetch. In scalar machines prefetches are usually done on a cache line basis, and a useless prefetch has a small penalty in terms of bus occupation and can be done while no other activity is going on in the system bus [CB92]. For a vector processor, a useless prefetch has a terrible cost both in terms of bus cycles and in terms of wasted buffer space.

The key for solving the latency problem in vector architecture is ensuring that the memory pipeline is busy at all times with useful data. If the memory ports are always busy, then a continuous stream of data is always flowing from the memory system into the processor, and, as a byproduct, bandwidth is fully used and memory latency is successfully hidden.

The major shortcoming of vector machines that goes against full usage of the memory pipeline is the simplicity of their in-order dispatching mechanisms. All major vector platforms execute vector instruction in strict program order. Although there is some overlapping between scalar and vector instructions a stall in a vector instruction typically prevents further vector instruction dispatching until the hazard is resolved [Rus78, Oed92, CGMW88, NKT$^+$95, WKI86] (see page 20 for further references).

The reasons for not improving upon this in-order issue scheme are unclear. Perhaps it has been thought that the inherent latency hiding advantages of vectors are sufficient. Alternatively, it is possibly because the first successful vector machine, the Cray-1, issued instructions in order, and additional innovations in vector instruction issue were simply not pursued.

1.6 DYNAMIC INSTRUCTION SCHEDULING

Dynamic instruction scheduling is a general term that refers to a wide variety of techniques that attempt at improving performance by on-the-fly reordering of a program's instructions. Several techniques have been devised to achieve this instruction reordering, but they all share in common that they look ahead in the dynamic instruction stream to find independent operations that might be executed in parallel.

The main contribution of this thesis is to show that by using dynamic instruction scheduling in a vector processor, the performance of vector programs can be greatly improved. Dynamic instruction scheduling reorders the computation stream to maximize the chances of having ready instructions that can be launched to free functional units. In particular, dynamic instruction scheduling can provide enough memory instructions to always keep the memory pipeline busy. This leads to having a continuous stream of requests flowing from the cpu to the memory system and a corresponding flow of data coming back from the memory system. If this flow is steady, then individual latencies are overlapped one with each other and the cpu does not experience delays.

We will present three different dynamic instruction scheduling techniques aimed at improving the performance of vector architectures. These techniques have been used in superscalar microprocessors and we will adapt them to vector architectures. The first two techniques, attempt at improving a single program performance by looking for independent operations in a single instruction stream. These two techniques are *Decoupling* and *Out-of-order execution*. The third technique, *Multithreading*, is orthogonal to the previous two and aims at finding independent instructions among several different programs. Its goal is to improve global throughput by interleaving different instructions from different programs at the same time.

Our goal is to present solutions that are fully binary compatible with existing vector instruction sets. Thus, the techniques described are fully transparent to the end user and do not require recompilation nor the addition of special opcodes to the ISA.

1.7 SPILL CODE

Spill code is a serious problem on architectures that have a limited number of registers at the ISA[1] level. The small number of logical registers available to the programmer or compiler induce extra traffic between the processor and the next memory level because when the compiler runs out of registers it must insert a store instruction to spill some register to the stack and, later, insert a load instruction to reload the data. Several situations can force the compiler to insert extra load/store instructions: First, large loops that compute more temporary values that can be held in registers force the explicit introduction of store and load pairs of instructions. Second, a register may be loaded from memory, but the data can not be saved for long because the register is needed for some other instruction. Then, when the memory data is needed again, it must be re-loaded. Third, variables that are reused between close iterations of a given loop are difficult to keep in the register file due to the high register pressure. Optimizations such as scalar replacement [CCK90] are hardly useful when the compiler has so few logical registers to use. Fourth, call return sequences, typically to intrinsic math routines, might involve saving and restoring of vector registers.

Traditional vector machines have had a relatively small number of vector registers (8 is typical). The successive Cray machines [Rus78, Fat89] and mini supercomputers such as Convex C3-Series [Con92] and Alliant [PM86] only have 8 logical vector registers. In this type of machines spill code can be very detrimental to performance due to two main reasons: first, when a vector register is spilled it involves a very large amount of data movement and, second, the distance, in terms of processor cycles, between a vector register and main memory can

[1] Instruction Set Architecture

be rather large. The limited number of vector registers was initially the result of hardware costs when vector register instruction sets were originally being developed; today the small number of registers is generally recognized as being a shortcoming.

The obvious solution to this register limitation is to add more physical and logical registers to a given machine. Alas, such a solution is not always feasible since binary compatibility if often more important than an increase in performance. Also, instruction sets that were not designed with an extension of the vector register file in mind might have opcode space problems to specify the extra registers. Even if the ISA extension is possible, old, dusty-deck codes might require a recompilation step to benefit from these extra registers, and this recompilation might not always be possible.

This thesis will also look at techniques aimed at dynamically eliminating the excess traffic caused directly or indirectly by the limited number of vector registers available. We will present three different solutions to the spill problem. The first one, based on the concept of a victim cache [Jou90] next to the vector register file, can be applied to any vector architecture. The other two solutions, are tailored to the out-of-order and decoupled vector machines we are proposing.

As we also pointed out for the dynamic scheduling techniques, our goal is to present solutions that are fully binary compatible with existing vector instruction sets. These spill elimination techniques will neither require special instructions nor will affect the user-level view of the processor.

It is worth noting that the work in spill code elimination presented is orthogonal to the dynamic instruction issue proposals. This spill code elimination is relevant in the context of machines having only 8 vector registers (such as Cray and Convex) but less important in the newer Japanese machines where either 32 logical registers are present in the ISA [EML88, KHG92] or the cpus have a two-level register file with 8 vector registers coupled to the functional units and additional reconfigurable storage (up to 128Kb) that can hold a variable number of vector registers (from 64 up to 2048 depending on the vector length) [Uch91, Wat91].

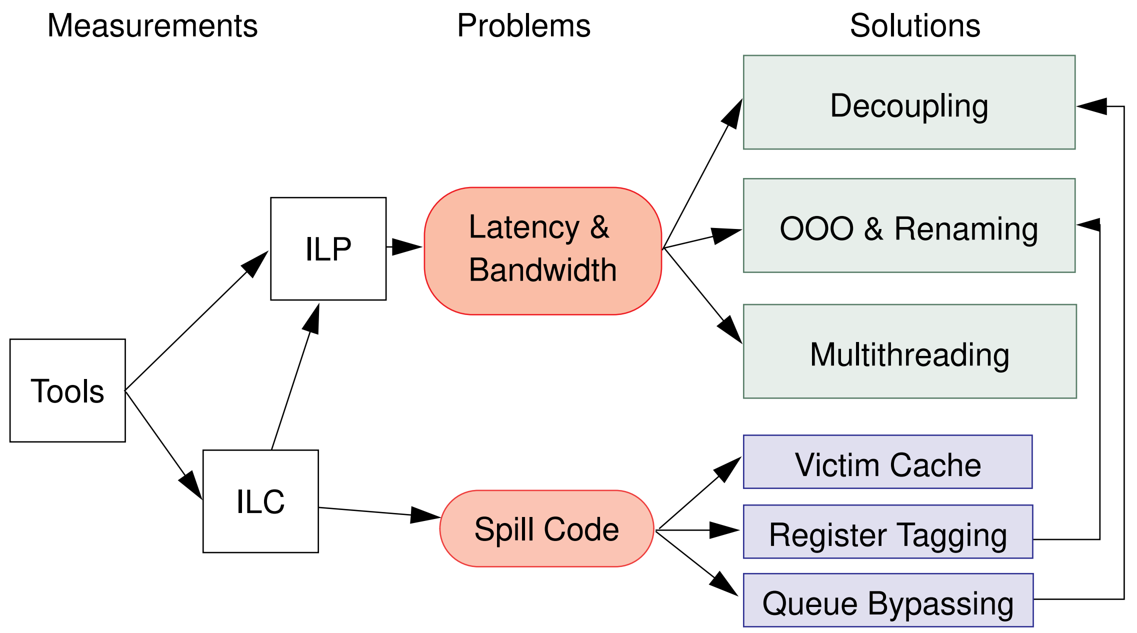

Figure 1.2 Outline of the topics presented in this thesis.

1.8 THESIS OVERVIEW

The general outline of the topics presented in this thesis can be seen in Fig. 1.2. All the work is based on simulation of a set of benchmarks running on a reference architecture and on the proposed enhanced architectures. The essential characteristics of the reference architecture are a single memory port, two functional units and 8 vector registers.

The first topic we deal with is a detailed analysis of the instruction level characteristics of a set of highly vectorizable codes (box ILC in Fig. 1.2). The second is an analysis of the instruction level parallelism available when these programs are executed on a traditional, in-order vector architecture (box ILP in Fig. 1.2). These two studies lead to two main observations:

- Most programs are memory bound when run on a vector machine having one memory port and two functional units. Despite this fact, the memory port can be idle for as much as 50% of the execution time of a program. Moreover, if memory latency is varied and, in particular, increased, the degradation in execution time is very significant.

- The amount of spill code present in the analyzed codes can be rather large.

The rest of the thesis deals with solutions to these two problems.

For the poor usage of the memory bus we present three solutions organized in two groups. The first group comprises of solutions that use dynamic instruction scheduling techniques that attempt to extract parallelism from a single program (this group includes two proposals: decoupling and out-of-order execution). The second group attempts to extract parallelism from the concurrent execution of several programs (multithreading).

The first proposal is to use decoupling techniques in the reference architectures. This first proposal aims at finding limited parallelism in the instruction stream by splitting computation and memory instructions. Nonetheless, this limited parallelism exploited is enough to make the resulting architecture latency tolerant and to effectively use all memory bandwidth.

The second proposal goes one step beyond decoupling. Instead of disordering memory and computation instructions only, we try to get maximum parallelism by allowing all types of instructions to execute fully out-of-order, only limited by true dependence constraints. To achieve this, we use an out-of-order execution engine coupled with register renaming. The benefits are manifold: on one hand, the architecture still splits memory and computation instructions, so a similar benefit to decoupling is possible. On the other hand, memory instructions themselves can be reordered and also computation instructions can be reordered. This effect maximizes functional unit usage, both in the memory and computational pipelines. Moreover, the mechanisms used for register renaming and out-of-order issue can also be used to support precise exceptions – generally a difficult problem in vector machines.

Another possibility to achieve a very good use of the memory port is to apply multithreading techniques. Here, instead of finding parallelism among instructions from a single program we interleave the execution of different programs so that when one of them blocks, another can use the available hardware resources. Each program by itself does not complete earlier, but a set of programs can be executed in a much shorter time than if they were run sequentially. Thus, the goal of this third technique is to improve global throughput at the expense of slowdowns of each individual program. By interleaving different memory instructions from different programs, our two goals are both achieved: first, memory bandwidth is not wasted, since it is very likely that there is always one pending memory instruction to be launched. Second, latency tolerance is successfully hidden. Although each individual program might see the same memory latencies as before, the global execution of several programs masks memory latency by executing several thread on otherwise unused resources.

For the spill problem, we also present three different solutions. Behind all of them, a single idea is present: to keep spilled variables in some form of storage space local to the processor. This way, when a spilled variable is later referenced, it will be retrieved from this local storage space without causing memory traffic and without paying main memory latency. The first solution, a victim cache, could be applied to any from of vector architecture. That is, it is both valid for in-order traditional vector architectures and for our three dynamically scheduled proposals. The second solution is tailored to the decoupled vector architecture and uses its hardware store queues to hold, for as long as possible, spilled data. The third solution, tailored to the out-of-order vector architecture, uses the renaming tables to keep spilled data in the processor. By artificially altering the rename tables, copies of vectors spilled to the stack can be kept in free physical registers and later retrieved with a very low execution cost.

1.8.1 Structure of this work

This work is organized as follows:

- Chapter 2 presents our working environment. The reference platform used for all simulations in this work is presented, as well as our set of benchmarks and tracing and simulation tools.
- Chapter 3 presents a detailed instruction level characterization of our benchmark programs, including distribution of operations, average vector length and percentage of spill code present in each program.
- Chapter 4 presents an analysis of the instruction level parallelism achieved when running the benchmarks on the reference machine. This analysis reveals the starting point for our work: memory bandwidth is not fully used.
- Chapter 5 describes a *Decoupled vector architecture*, our first proposal for solving the bandwidth and latency problem. We also present a technique to dynamically eliminate spill code using the hardware queues present in a decoupled architecture.
- Chapter 6 describes an *Out-of-order vector architecture* with register renaming, our second proposal for solving the bandwidth and latency problem. This chapter also deals with precise exceptions and presents a register tagging technique that dynamically eliminates memory traffic.

- Chapter 7 describes a *Multithreaded vector architecture*, with the aim of increasing total throughput and maximizing memory port usage.
- Chapter 8 presents a general spill code elimination technique, not tied to any of the particular implementations described in chapters 5–7, based on the concept of a victim cache for vectors next to the vector register file.
- Finally, chapter 9 summarizes the contributions of this thesis and presents open areas for future research.

2

PLATFORM, TOOLS AND BENCHMARKS

Summary

This chapter presents the reference machine on which this thesis has been carried out, the Convex C3400. We describe the basic functionality of the machine and the software tools necessary to gather traces of vectorizable programs that run on it. We also discuss the simulation tools necessary to study the performance characteristics of programs on a general vector architecture. We introduce our benchmarks and discuss the necessity of highly vectorized benchmarks. Finally, a simplified model of the C3400 is presented that will be the base of all further comparisons. This model is referred to as the "reference architecture".

2.1 INTRODUCTION

This study is based on a traditional vector processor and numerical applications, primarily because of the maturity of compilers and the availability of benchmarks. We have used trace-driven simulation as the base of our experimentation methodology. All the proposals presented in this work have been evaluated by comparing the performance of a trace driven execution of a reference machine against the performance of each individual proposal.

2.2 WHAT IS A TRADITIONAL VECTOR ARCHITECTURE ?

The definition of a "traditional" vector processor is necessarily vague, since, as appendix A shows, the variation in computational power and memory bandwidth of past and present machines is rather large. Nonetheless, there are several key points that can identify what we consider a traditional vector architecture:

In order execution : The most clear feature of traditional vector architectures is an in-order execution model of vector instructions. At the time this study was started, all major vector platforms executed vector instruction in strict program order. Although there was overlapping between scalar and vector instructions a stall in a vector instruction prevented further vector instruction dispatching until the hazard was resolved. Examples of these behavior are found in most past and present parallel vector processors. Cray machines, starting with the Cray-1 [Rus78], and the follow-on's, the Y-MP [Cra90, VSH91, Vaj91] and the C90 [Oed92, OW93] execute vector instructions in strict program order. The Convex C-series machines (the C2, C3 and C4 series [CGMW88, Con92]) behaves in a similar way. Some machines do actually have parallel instruction issue in the scalar unit, such as the Fujitsu VPP300 and VPP500 [NKT+95]. These processors have a long instruction word (LIW) format that allows three different operations to be packed into a single instruction. However, the vector instructions are executed in-order. The NEC SX-2 [WKI86, Fat90] and SX-3 [Wat91, IW91, WI92] also execute vector instructions in order. Only very recently, the new SX-4 [NEC95, GBH96] has been implemented using out of order execution in the vector and scalar pipelines [Tan96].

Register-to-Register : All current vector machines are register-to-register architectures and only a few early vector machines, the CDC Star-100 [HT72] and the TI-ASC [Wat72], their successors, the Cyber-205 [Arn82, Arn83, HMD+86] and ETA-10 [Fat89] did have memory-to-memory architectures. Since the register-to-register model has many clear advantages, as its commercial success shows, we will discard beforehand any further study of memory-to-memory machines. Among the register based machines, we can clearly distinguish between the two-level machines, mostly Japanese supercomputers, and the single level machines (Cray, Convex, Alliant). Single-level machines have a limited number of vector registers connected through a crossbar (or a restricted crossbar) to the functional units. Cray, Convex and Alliant machines all have 8 logical vector registers each holding 64 (Cray Y-MP) or 128 words (Cray C-90, Convex). Two-level machines have a reduced set of vector registers (typically 8) connected in a very limited way to the functional units and then provide a large second level set of registers. In order to operate on data, it must be moved from main memory to the second level registers and then down to the first level. Final program results proceed in the opposite way, being copied from first level to second level and then to main memory. The second level is typically variable in size; that is, its total storage capacity is fixed (N words) and it can be split in a certain number of logical registers (R registers) each of which holds M words of data. The choices for R and M are left to the programmer and/or compiler, with the obvious restriction that $N = R \times M$, and that, usually, R and M must be powers of 2. An example of such machines is the NEC SX-3 with 128Kb of second level vector register storage, where $N = 16Kwords$ and where R can be 64, 128 or 256. The Fujitsu VP2600 [Uch91, MNT91, TOU92] has a similar configuration.

Memory-to-Computation Balance : We can take the different vector architectures described in appendix A and compute their peak floating point capacity and their peak memory-to-register bandwidth. Then, the ratio of these two numbers gives us an intuition of the balance in such machines. For example, for the Cray-1, with 2 floating point units and 1 memory port, we get a ratio of 0.5. This ratio indicates, in a very informal way, that if the programs *do not* have, on average, twice as many computations than memory transfers, the memory port is very likely to be the execution bottleneck. Table 2.1 summarizes these ratios for several machines. Column "Flops/Cycle" indicates peak number of floating point operations that can be performed per cycle. Column "Words/Cycle" indicates peak number of 64-bit words that can be transferred per cycle. The notation $L + S$ (in the Cray Y-MP, for example) indicates L words can be loaded and S words can be stored per cycle.

Most machines have almost double computational capacity than memory transfer capacity, which is not surprising given current technology constraints. It is relatively easier to increase the number of functional units than it is to increase the number of memory ports (or their width). Only the X-MP lineage (that is, the X-MP, the Y-MP, the C-90 and the T-90) seem to present the inverted situation. From a cost point of view, ratios over 1

Machine	Year Introduced	Flops/ Cycle	Words/ Cycle	Ratio
Cray-1	1976	2	1	0.5
Fujitsu VP100	1982	3	2	0.66
Cray X-MP	1983	2	2+1	1+0.5
Cray-2	1985	2	1	0.5
Cray Y-MP	1988	2	2+1	1+0.5
Fujitsu VP2600	1990	16	8	0.5
NEC SX-3	1990	16	8+4	0.5+0.25
Cray C-90	1992	4	4+2	1+0.5
NEC SX-4	1996	16	16	1

Table 2.1 Balance of peak computation rate and peak memory transfer rate for several vector machines

do not make much sense in today's technology. Given a certain budget, it should always be possible to have a ratio lower than 1, since the memory system is usually responsible for a high fraction of overall system cost.

In our study we take the simplest machine having a ratio of 0.5: a machine with 1 memory port and 2 functional units that can deliver two flops per cycle. We believe this choice will be representative of the broad range of machines having similar ratios.

2.3 PLATFORM: CONVEX C3400

The platform chosen for this study, a Convex C3400 [Con92], has the three major features described in the previous section: it executes in-order, has a register-to-register architecture with a single register level and has a compute-memory ratio of 0.5.

2.3.1 Hardware Description

A single processor of a Convex C3400 consists of a scalar unit and an independent vector unit connected through a single memory port to an interleaved memory system.

The architecture defines three different sets of registers: A, S and V. Registers of type A, the *address* registers, are used to generate the base address for all memory references. Thus all types of load/store instructions use at least one A-register in their addressing modes. A-registers are 32 bits wide. The *scalar* registers, S-registers, are 64 bits wide and are used to carry out all floating point computations in scalar mode. Also, S-registers are used in vector instructions whenever a vector is operated with a scalar value. The *vector* registers, V-registers, hold 128 64-bit words and are used for all types of vector computations, both integer and floating point.

Scalar Unit

The scalar unit executes all instructions that involve scalar registers (A and S registers), and issues a maximum of one instruction per cycle. There are 8 A-registers and 8 S-registers, named A0 through A7 and S0 through S7. Registers A0, A6 and A7 are used by the compiler as the stack pointer, the argument pointer and the frame pointer, respectively. All references to the run-time stack are usually made using A7, and, in particular, all register spilling is done using A7 as the register to compute base addresses.

There is a 16Kb data cache that holds scalar data only that is kept coherent with the vector reference stream by serializing all memory references (scalar and vector) through a single port. Vector stores invalidate the necessary lines in the data cache so that subsequent scalar memory accesses are forced to reach main memory to get the most up to date value. Vector loads never read the scalar cache.

Vector Unit

The vector unit consists of two computation units (`FU1` and `FU2`) and shares the memory accessing unit (`MEM`) with the scalar part. The `FU2` unit is a general purpose arithmetic unit capable of executing all vector instructions. The `FU1` unit is a restricted functional unit that executes all vector instructions *except* multiplication, division and square root. Both functional units are fully pipelined.

Vector operations are made under control of the *vector length* (VL) register. This scalar register, 8 bits wide, indicates the desired length, from 0 to 128 elements, of vector operations. We note that values of 0 are possible, and do actually occur when the compiler generates code to dynamically compute a certain vector length. Also note that the VL register behaves in "truncation mode": if the value being assigned to VL is greater than 128, then the hardware automatically sets VL to 128. Vector memory operations use a second control register, the *vector stride (VS)* register. This register indicates the stride, in bytes, that separates the bytes that must be fetched from memory. Finally, the *vector mask (VM)* register is a 128-bit wide register that allows all vector operations to be done under the control of a mask. Each bit in the VM register determines if individual operations of a vector instruction are executed or not.

The vector unit has 8 vector registers which hold up to 128 elements of 64 bits each. The eight vector registers are connected to the functional units through a restricted crossbar. Pairs of vector registers are grouped in a register bank and share two read ports and one write port that links them to the functional units. The compiler is responsible for scheduling vector instructions and allocating vector registers so that no port conflicts arise.

The C3400 implements vector chaining from functional units to other functional units and to the store unit. It does not chain memory loads to functional units, however (nor do the Cray-2 and Cray-3 [Fat89]). Although such chaining could be done, it is more complicated than other chaining because the memory system may not deliver the individual vector elements in order. We note that the Convex compiler used for this study takes the lack of load chaining into account when scheduling vector instructions.

The datapath is designed so that 32-bit values can be processed at twice the speed of 64-bit values. The vector unit can be producing 2 64-bit results each cycle or 4 32-bit results per cycles using the available width of the datapath.

The vector unit is clocked at 40ns, but the divide unit is clocked somewhat faster, at 13.3ns. This compensates for the fact that the divide unit is not able to produce one 64-bit result per cycle.

Memory Subsystem

The C-3400 memory system is organized in 4 sections with 8 memory modules in each section. Each module has an access time of 320ns, which represents 8 processor cycles (the vector cpu cycle is 40ns). Every cycle a 64-bit load or store can be executed, except when memory accesses are misaligned, in which case it would take 11 cycles to complete a memory transaction. The memory system is interleaved so that sequential addresses would visit first all modules in a section and then change section. Since each memory module holds a 64 bit element, smaller memory accesses (32 bits) can proceed at larger speeds (if using stride 1) by transferring two words using the 64 bit datapaths.

The interconnection network between cpu and memory is a 5 x 4 crossbar (in an 8 processor system) where every two processors share a port to memory and there is a fifth port for physical I/O. Note that for the purposes of this thesis, a cpu has a full dedicated memory port, since we do not model the fact that this memory port would be shared by two processors in the real machine.

2.3.2 Compiler Description

The compiler used to compile all benchmarks has been the Convex FC version V8.0. All codes have been compiler using the -O2 flag, that includes all scalar optimizations plus vectorization and vectorizing optimizations. The next optimization level (-O3) implied parallelization and thus was not used. All binaries produced were fully linked, that is, no shared libraries where used.

The compiler was responsible for scheduling all vector instructions and allocating vector registers. One of the most important optimizations that it performs is the careful scheduling of the register file ports. Since each pair of vector registers (V0 and V4) only has 1 write port and 2 read ports, the order in which the register banks are referenced is very important.

The quality of the vectorized code is good in general and the vectorizer performs relatively well on most vectorizable loops [CDL88]. The major limitations that could be mentioned are that the resulting scalar code usually has many spilled variables and that loop unrolling was almost impossible to apply due to the low number of vector registers.

2.4 REFERENCE ARCHITECTURE MODEL

Although the C3400 fits quite well our informal definition of a "traditional" vector architecture, it does have too many technological constraints that do not make much sense in today's technology. Moreover, some of the functional unit latencies and some issue hazard conditions would be obsolete today. Since all our study is based on simulations, it is important to ab-

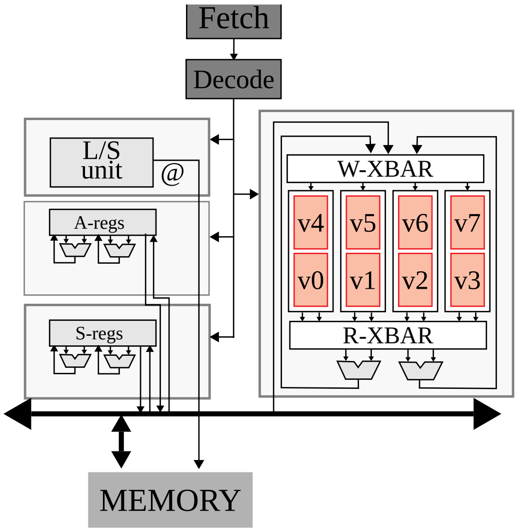

Figure 2.1 The reference vector architecture.

stract a model from the C3400 that retains the major features of a traditional architecture but ignores low level details. As an example, the C3400 implementation needs 7 cycles from the point at which the VL register is set to the point where the following vector instruction can be issued. Such a performance penalty, while important for real codes running on the real machine, should not be included in our study. It would obscure too much our results and could hide other potential problems of the machine.

Instead of using a detailed model of the C3400, we want an idealized machine that has as few special cases and particular technology dependences as possible and while reflecting the problems of a wide variety of machines. The result is a new architecture, that we refer to as the *reference architecture model*, with the following characteristics (see Fig. 2.1):

- A scalar unit having two functional units for A-type computations and two functional units for S-type computations. The A and S register files have enough ports to satisfy all functional unit demands.

  Rationale: this scheme is closest to today's microprocessors.

- A vector unit consisting of two asymmetric functional unit (same as C3400). Unit FU2 performs all types of vector computations while unit FU1 performs only add-like operations.

Rationale: The divide and square root functional units are costly in terms of processor area and these two operations are not as frequent as multiplications and additions. Most processors, both micros and vector machines, make tradeoffs and have more units devoted to multiplication and addition than division and square root.

- A single memory port on which all types of memory references are made. Memory instructions are always executed in strict program order.

  Rationale: Since memory is the most expensive component of a vector machine, it is reasonable to unbalance the machine to have more functional units than memory ports to ensure that the latter will be busy for many cycles.

- Memory is considered as a black box that delivers results with a latency of L cycles. No sections or bank conflicts are modeled and, thus, all vector loads, stores, gathers and scatters proceed at the same speed.

  Rationale: This greatly simplifies the analysis of the behavior of the architecture under different memory latencies.

- 8 vector registers, organized in 4 banks, each bank having 2 read ports and 1 write port.

  Rationale: Although this approach is not very common, we keep this structure because the compiler goes to great lengths to re-schedule the code to efficiently use the register file ports. Another alternative would have been to have 8 registers each with 1 read and 1 write port, as the Cray machines, but that could have caused too many port conflicts given the Convex scheduled binaries. The third alternative, having 8 registers each with 2 read ports and 1 write port was considered too good and that could provide too much of an advantage.

- Uniform functional unit latencies for all instructions, shown in table 2.2. These latencies are fixed and do not depend on types of operands or on pairing of instructions, as is the case in the C3400 implementation. The values chosen are not exactly those of the C3400, but reflect current trends in microprocessor design and vector design.

  Rationale: The original C3400 latencies are out of date and reflect many of the technology constraints of a low-cost ECL implementation.

- Uniform processing speed for all vector instructions, regardless of operation on 32 or 64 bits. All vector units are considered to be fully pipelined and a new operation can be started every cycle.

  Rationale: different speeds would make the results even more dependent on benchmark characteristics. Most vector architectures have a single speed, whether data is 32 or 64 bits. Although this is an interesting field of study for new application domains, such as multimedia, for the purposes of this thesis having a uniform processing rate simplified all following analysis.

Parameters	Scalar (int/fp)	Vector
read x-bar	–	2
write x-bar	–	2
vector startup	–	1
add	1/2	6
mul	5/2	7
logic/shift	1/2	4
div	34/9	20
sqrt	34/9	20

Table 2.2 Latency parameters for the reference architecture model.

- Scalar memory accesses are considered to always hit in the data cache, which is only 2 cycles away from the processor. This does not mean that they can be done in parallel with vector memory operations. A scalar memory access has to wait for all prior memory operations to be completed before accessing the data cache.

  Rationale: The percentage of scalar memory accesses in our benchmarks is very low and they do not usually occur in vector sections of the code. The difference of modeling real hits and misses would be very small and would substantially slowdown all simulations, since vector stores would have to invalidate references present in the data cache.

- Fetch and Decode bandwidth: 1 instruction per cycle.

  Rationale: Since each vector instruction specifies many operations, achieving a high ILP is possible without needing parallel instruction fetching and/or decoding. Even if vector instructions are dispatched on successive cycles, after a small startup phase it is possible to have several operations being executed each cycle.

- 100% hit ratio in I-cache.

  Rationale: The I-cache footprint of the vector programs is very small. Execution time is mostly concentrated on a few, not very large, loops. Therefore, modeling the I-cache would introduce too much extra cost in terms of simulation time without providing any additional insight into the memory latency problem under study.

- Due to lack of tracing facilities, system calls invoked through the `sysc` instruction are treated as a single cycle operation. Thus we are not modeling any of the code executed inside the operating system.

2.5 SIMULATION ENVIRONMENT

As already mentioned, we have used simulation as the base of our experimentation methodology. The advantages of trace driven simulation are the ease of reproduction of results and the highly controlled environment in which we execute the experiments. Since the trace is fixed and stored, different simulation runs can not be altered or influenced by external factors such as machine load or memory conflicts. Once the behavior of a set of programs has been captured on the trace, the execution time of the program can be approximated on a variety of different configurations of machines.

We will now briefly describe the software tools that we have developed in order to perform the necessary simulations. The first tool, a tracing tool, has been targeted to a C3400 and is able to gather traces of binaries running in vector mode on a single processor of a C3400. The second tool is, in fact, a set of different simulators: we have developed a simulator for each one of the architectures under study: the reference architecture, the decoupled architecture, the out-of-order architecture and the multithreaded architecture. Also, an extension of the reference architecture that has a victim cache has been developed.

2.5.1 Tracing tool: Dixie

All data presented in this study has been gathered using trace-driven simulation. We have developed a pixie-like tool called Dixie [EM94] that is able to produce a trace of basic blocks executed as well as a trace of the values contained in the *vector length* (`VL`) register. The ability to trace the value of the *vector length* register is critical for accurate detailed simulation of the program execution, because each vector instruction can execute potentially with a different vector length. Thus, our measurements do not suffer from the problems reported in [VSH91] and [RR94]. It is also important to note that Dixie instruments the output of a commercial compiler without requiring any special properties in the binary, and that this tracing method gives a high precision in all of our measurements.

The tracing procedure is as illustrated in Fig. 2.2: the benchmark programs are compiled on a Convex C3400 machine using the Fortran compiler (version 8.0) at optimization level -O2. Then the executables are processed using Dixie, which decomposes the executables into basic blocks and instruments the basic blocks to produce four types of traces: a basic block trace (BB trace), a trace of all values set into the *vector length* register (VL trace), a trace of all values set into the *vector stride* register (VS trace) and a trace of all memory

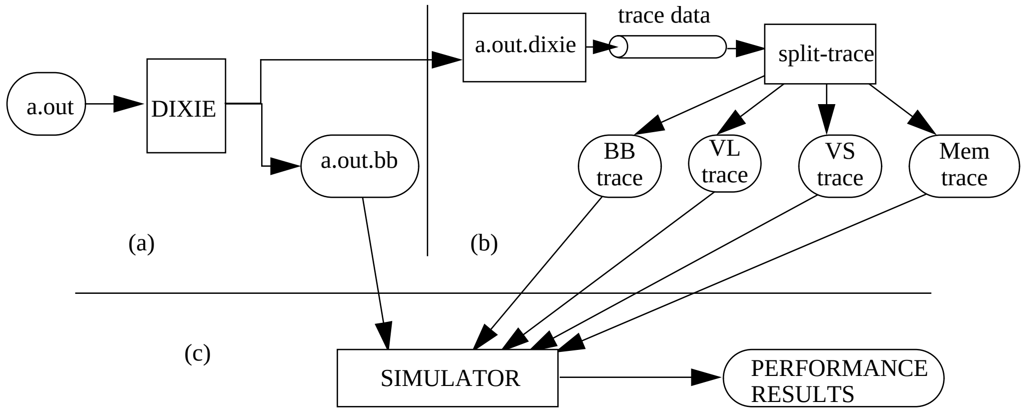

Figure 2.2 The instrumentation process. Step (a) consists in processing a program's executable and generating an instrumented version of it. In step (b) we run the modified executable on the Convex C3 machine and we obtain a set of traces that fully describe the execution of the program. In (c) this set of traces is fed into the simulator, which will do a cycle-by-cycle execution of the program and will gather performance results.

references (actually, a trace of the base address of all memory references) (Mem trace). Dixie instruments all basic blocks in the program, including all library code. This is especially important since a number of fortran intrinsic routines (SIN, COS, EXP, etc.) are translated by the compiler into library calls. These library routines are highly vectorized and tuned to the underlying architecture and can represent a high fraction of all vector operations executed by the program. Thus it is essential to capture their behavior in order to accurately model the execution time of the programs.

Once the executables have been processed by Dixie, the modified executables are run on the Convex machine. These runs produce the set of traces that accurately represent the execution of programs.

2.5.2 Parameterizable simulator: Jinks

Jinks [Esp95] is a parameterizable simulator that implements the reference architecture model described in section 2.4. In reality several simulators are folded into a single simulator (reference, decoupled, out-of-order and multithreaded architectures). Jinks is implemented in around 30,000 lines of C code.

The traces gathered using Dixie are then fed to Jinks. Jinks closely models all dependences found in the execution of a program on the different architectures. It does cycle-level simulation of the reference architecture and keeps detailed information about resource usage, number of hazards produced, stop issue conditions etc.

2.6 BENCHMARK SET

A very important choice in this study is the set of programs to be analyzed using simulation. We have chosen numerical applications as the source for our benchmarks because these are the types of applications that have been run traditionally on vector machines. The main characteristic of these applications is that they tend to be dominated by do-loop style computations that consist of several numerical expressions.

As a starting point, we took the Perfect Club programs [Ber89] and the Specfp92 [CKDK91] programs. The Perfect Club codes were a joint effort of the CSRD at the University of Illinois, several supercomputing vendors, several universities and third party software houses, to put together a set of codes representative of the workloads of large machines, including vector supercomputers. Thus they represent an ideal target for our study. Their main drawback is that, in order to make them manageable, their data sets where somewhat trimmed down. While this is not a problem for our present study, it makes these programs less useful in the context of cache based systems. The Specfp92 codes are today in widespread use for evaluating the performance of workstations. Although they are not specifically targeted at vector machines, they contain a fair amount of applications belonging to the scientific and engineering domains that have also been traditionally run on vector machines.

The second consideration in choosing the benchmarks is that they should be representative of the class of codes that are actually run on real vector machines. This implies that the programs should be fairly vectorizable and should run relatively well on a real vector machine. This thesis is *not* about the optimization of general purpose programs but on the optimization of vectorizable codes. It does not make sense, in the context of this thesis, to study programs that have only a 2 or 3% vectorization. These types of programs should not be executing on a vector machine in the first place, and, second, will not experience any noticeable improvement from our techniques. *It is very important to stress the point that we are not studying optimizations that will be useful for all types of programs. Rather, we are seeking optimizations that will*

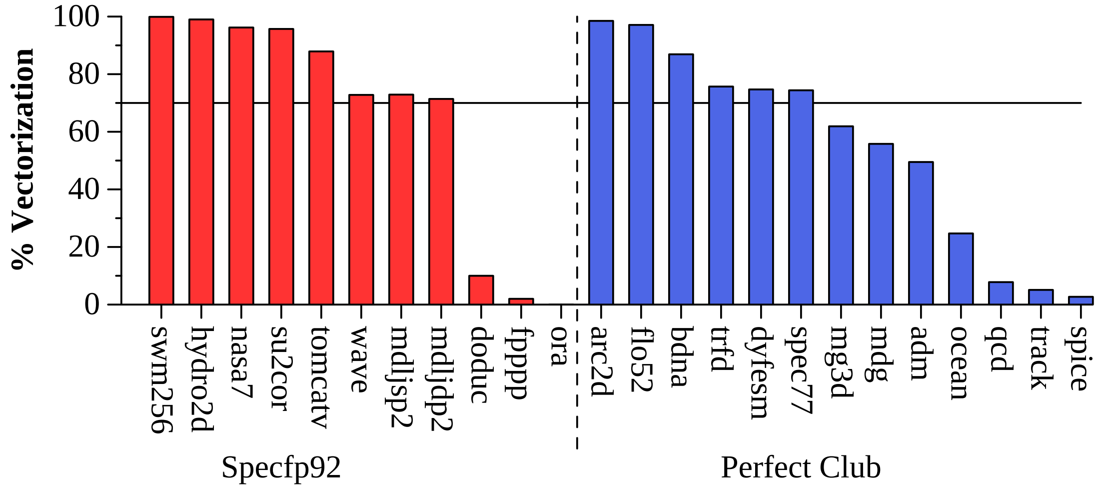

Figure 2.3 Percentage of vectorization of programs from the Perfect Club and Specfp92 suites.

make those programs that already run well on vector architectures run at their maximum possible speed.

In order to evaluate these second criteria, all programs from the Perfect Club and Specfp92 were compiled on the C3400, and their traces gathered using Dixie. Then, using the detailed information in the traces we computed the percentage of vectorization of each program as follows:

$$\frac{Operations\ executed\ in\ vector\ mode \times 100}{Operations\ executed\ in\ vector\ mode\ +\ Operations\ executed\ in\ scalar\ mode}$$

We use the term *operation* rather than *instruction* because each vector instruction performs several operations (from 0 to 128) while each scalar instructions performs exactly 1 operation. Figure 2.3 presents the results for the analyzed programs.

We decided to select 10 programs that had a percentage of vectorization over 70%. Once again, the rationale is to study those programs that are supposed to perform very well on vector machines and see if (a) it is really true that they achieve a sustained performance close to peak and (b) see if they can be improved using dynamic scheduling techniques. The selected programs were:

From the Specfp92 : `swm256`, `hydro2d`, `nasa7`, `su2cor` and `tomcatv`.

From the Perfect Club : `arc2d`, `flo52`, `bdna`, `trfd` and `dyfesm`.

These 10 programs will be used throughout the rest of this thesis to evaluate the performance of the reference machine as well as to compare the performance of our main three proposals.

3

INSTRUCTION LEVEL CHARACTERIZATION

Summary

This chapter studies the instruction level characteristics of the selected Perfect Club and Specfp92 programs when compiled and executed on the reference platform. Using a trace driven approach this chapter presents data on the degree of vectorization of the programs, the vector length used in vector instructions, the operation type distribution and the balance between memory and computation operations. We also study the spill code introduced in the program by the compiler and the pressure on the dispatch unit of the vector architecture.

3.1 INTRODUCTION

Vector architectures have historically been evaluated by measuring overall performance on benchmark programs. For example, one might run benchmarks like the Perfect Club [Ber89] suite or Linpack [Don93], and then summarize overall performance in MFLOPS or total elapsed time. Many researchers have looked at the performance of engineering and scientific codes running on vector supercomputers [IMS88, NPR88, SW88, YZ91, SWL+92]. While this method certainly provides some interesting information for the application user, it does not provide direction to the architect or designer of vector machines nor does it provide any clear reasons why one program may perform much better or worse than another.

Before analyzing performance characteristics of a set of benchmarks on a certain machine, it is very important to have a detailed knowledge of the low level characteristics of such benchmarks. In order to be able to reason on performance deficiencies of a program it is necessary to know in detail the resource usage made by the dominant parts of the program. Without this resource usage knowledge it is very difficult to determine whether a performance problem could be easily solved by adding some extra functionality or whether the problem is more complex and requires a significant amount of work.

As an example, consider looking at the performance of a vector loop on our reference architecture. The number of iterations of the loop, and, hence, its vector length, will be key in understanding the impact of functional unit latencies on its performance. Furthermore, knowledge of its ratio of memory accesses to computations can help to analytically determine performance limits that are intrinsic to the loop and that can not be improved. Thus, if a loop has 20 memory operations and 10 computations we know that regardless of execution model, our reference machine will not be able to execute it in less than 20 time units (a unit being the time necessary to access a piece of memory). Only after these performance bounds are known it is useful to start looking at the actual performance of the loop. If it is close to its limit, there is little improvement to be made (unless, of course, one augments the available resources, but that would be the topic for another thesis). If it is far away from the limit, then the dynamic scheduling techniques being studied in this work can be of help.

Such kinds of low level studies have been used for guiding machine implementations for some time, and this approach became popular in the early 80s as part of the RISC philosophy of microprocessor design [HP90]. For example, the detailed knowledge of the relative frequency of execution of instructions [CB94b, ML93] has led to the inclusion of new compound operations (like the fused multiply add instruction) in the instruction set so that the "common case" is optimized and executes in fewer cycles. While this detailed approach has been routinely applied by the microprocessor design community, there are very few published studies of this kind in the vector computer arena.

This chapter will look at the low level characteristics of our set of vector benchmarks. In particular we will be interested in the following measurements:

Instruction and Operation Counts : these allow analysis of vectorization degree and pressure on the fetch and issue units. Also, knowledge of the distribution of floating point vector operations helps in analyzing the necessary balance inside the vector processor.

Vector Length Distribution : An accurate measurement of the vector lengths used in vector computations is crucial to understand the interaction between latencies and performance. Larger vector lengths help in amortizing all kinds of latencies, and, in particular, memory latencies. Moreover, the vector length influences the pressure on the fetch and decode unit, since the larger the vector length, the lower the number of instructions that must be decoded and executed.

Ratio of Memory and Computations : This is a key information for this thesis. The relation between memory accessing instructions and computation instructions inside inner loops determines the resource that will be the bottleneck when the loop is executed. Thus data showing this ratio for all the basic blocks containing vector code found in our benchmarks is needed to demonstrate that the majority of the loops are memory bound on a single port vector computer.

Spill Code : As already mentioned, spill code will also be studied in this thesis. This chapter will present data on the amount of spill code present in the benchmarks.

3.2 INSTRUCTION AND OPERATION COUNTS

Table 3.1 presents some basic statistics for the ten benchmark programs. The first column in this table contains the total number of basic blocks (in millions) executed by each program. The next two columns contain the total number of instructions issued by the dispatch unit, broken down into scalar and vector instructions. In this study we will distinguish between *instructions* and *operations*. A scalar instruction performs a single operation, while a vector instruction performs a varying number of operations, depending on the value of the vector length (`VL`) register. Column four presents the number of operations performed by the vector instructions (note that the scalar operation counts are identical to the scalar instruction

Program	#bbs	#insns S	#insns V	#ops V	% Vect	avg. VL	Scale Factor
`swm256`	2.0	6.2	74.5	9534.3	99.9	127	.31
`hydro2d`	4.8	41.5	39.2	3973.8	99.0	101	.13
`arc2d`	5.2	63.3	42.9	4086.5	98.5	95	.14
`flo52`	5.7	37.7	22.8	1242.0	97.1	54	.04
`nasa7`	17.1	152.4	67.3	3911.9	96.2	58	.13
`su2cor`	22.9	152.6	26.8	3356.8	95.7	125	.11
`tomcatv`	19.9	125.8	7.2	916.8	87.9	127	.03
`bdna`	47.0	23.9	19.6	1589.9	86.9	81	.05
`trfd`	44.8	352.2	49.5	1095.3	75.7	22	.04
`dyfesm`	34.5	236.1	33.0	696.2	74.7	21	.02

Table 3.1 Basic operation counts for the 10 benchmark programs.

counts). The fifth column is the percentage of vectorization of each program. We define the degree of vectorization of a program as the ratio between the number of vector operations and the total number of operations performed by the program (i.e., column four divided by the sum of columns two and four). Column six presents the average vector length used by vector instructions, and is the ratio between vector operations and vector instructions (columns four and three, respectively). All tables that will be presented in this study have been sorted following the degree of vectorization column. Finally, the last column indicates a scale factor to be used throughout this chapter when averaging results from the benchmarks. This scale factor is found by dividing the number of operations in each benchmark by the total operations for all the benchmarks. By using scale factors in averages, the averages will reflect a contribution from each benchmark as if the same number of operations were performed by each. This is necessary because of the very wide difference in the numbers of operations in the benchmarks. For example, `swm256` is almost twice as big as the other programs and using an unweighted average would cause it to dominate the averages unnecessarily.

The first interesting point from table 3.1 is the average vector length observed in programs (section 3.4 will present more detailed data about vector lengths). Even for these highly vectorizable programs, the average vector length varies a lot. Note that the C3400 has a maximum vector length of 128 elements. Only four programs, `swm256`, `tomcatv`, `su2cor` and `hydro2d`, behave as one would expect from highly vectorizable codes having the average vector length very close to the maximum register size. The rest have substantially smaller vectors.

In [Ber89] it is reported that the data sets distributed with the Perfect Club benchmark are not very large in order to allow these programs to run on a wide variety of computers, ranging from workstations to supercomputers. This may explain why the observed vector lengths for the five codes belonging to this suite are small, since a data set suited for a vector supercomputer would take too long to be run on a workstation.

The average vector length is very important when long memory latencies are considered. For example, assume we would like to extract maximum performance from a vector memory system having 80 cycles of memory latency. If we run on such a machine program `trfd`, around 4 different memory loads (requesting each one 20 memory items on average) would be needed to cover those 80 cycles of memory latency. The relationship between vector length and memory latency will play an important role in how much a vector architecture can tolerate large memory latencies.

In table 3.2 we have included the basic operation distribution for the 10 benchmarks. We have classified all operations executed by each programs in two major groups (Vector and Scalar) and each group has been further broken down into a total of four categories. These four categories are: `Load`, `Store`, `Mul` and `Add` operations. In the vector group, the `Load` and `Store` categories include, respectively, the `gather` and `scatter` instructions. The `Mul` category includes all types of instructions that can only be executed on the general purpose functional unit, that is, includes all multiplications, divisions and square roots. The `Add` category includes all other arithmetic operations not included in the `Mul` category, that is, adds, subtracts, logicals, shifts, etc.

What can be seen from table 3.2 is that all programs under study are likely to be memory bound when run on an architecture like the C34, having only one memory port and two functional units. Since we only have one memory port, all operations listed under columns `Load` and `Store` will use the same hardware resource, while operations listed under columns `Mul` and `Add` can *potentially* run in parallel (note that we emphasize the word "potentially", since the distribution of these two types of operations might in some cases prevent parallel execution).

Also, the total number of memory accesses performed in scalar mode is not very high. The majority of the scalar instructions are found in the `Add` category. This is not surprising, since scalar code found in vector loops is mostly composed of address computations (additions and shifts) and loop control instructions (decrements and comparisons).

	Vector				Scalar			
Program	L	S	*	+	L	S	*	+
swm256	3154	1345	2046	2990	0.1	0.1	0	4
hydro2d	1318	452	630	1573	11	3	1	22
nasa7	1069	652	1071	1119	52	21	3	62
su2cor	988	507	926	936	42	24	0.4	66
tomcatv	338	176	176	228	13	0.3	0	92
arc2d	1365	565	1247	909	23	6	1	29
flo52	469	222	229	322	10	5	0.3	18
bdna	408	292	383	506	66	29	29	99
trfd	433	224	217	221	122	47	1	152
dyfesm	290	108	150	149	75	30	2	106

Table 3.2 Balance between vector and scalar operations (all columns $\times 10^6$).

3.3 DETAILED OPERATION BREAKDOWN

To better understand the distribution of operations performed by our codes, we have further classified all vector operations into 11 subsets. Table 3.3 presents this classification. We first classify all vector operations as being either memory or compute operations. The compute operations appear in columns 2 through 8 in table 3.3, and the memory operations appear in columns 9 through 12. The first column of the table is the total number of operations (in millions) performed by vector instructions (and is the same as the fourth column in table 3.1). All the other columns are percentages taken over this first column, and thus, columns 2 to 12 add up to a 100%[1].

In the compute category, we have distinguished between seven types of operations. The two most common operations are additions and multiplications (subtractions are included in the "+" category). As can be seen, almost all programs have 90% of compute operations split between additions and multiplications, except for program `hydro2d`, where we found a large number of operations under the "other" category. We investigated this program, a hydrodynamical Navier-Stokes equation solver, to find the source of these operations in the "other" category. We found that the program spends most of its time in routine FILTER where anti-diffusion fluxes are limited. this routine is composed of a set of loops that compute differences

[1] Entries in the table have been rounded to maximize readability, so there might be some minor inaccuracies when adding up a whole row.

in the Z- and R- using "IF's" and "OR's" followed by a set of loops where MAX/MIN computations are carried out to actually limit the fluxes. Both the IF's, OR's and MAX/MIN make heavy use of `mask`, `or` and comparison instructions.

Overall, additions and multiplications appear to be approximately in 1 to 1 proportion. This suggests that in order to have a well balanced architecture (at least in terms of functional units), the number of units able to perform a multiplication and the number of units able to perform add-like operations should be the same.

The other types of compute operations are much less frequent. Nevertheless, it is very interesting to see in table 3.3 the percentages for high-cost operations like division and square root. Usually, hardware implementations of division and square root are not fully pipelined, and thus, may require a high number of cycles between consecutive dispatching of the same operation. For example, consider that a square root requires 20 cycles to complete and that it is not pipelined (that is, a second square root operation will have to wait the full twenty cycles before starting). Then, for example, the importance of the 1% square root operations performed by `bdna` is amplified – perhaps by as much as 20 times if a series of square roots are forced to operate serially.

Under columns labeled $\prod$ (`prod`) and $\sum$ (`sum`) we have included the frequency of operations performed using the reduction operators available in the C34 architecture. In terms of operation complexity, the `prod` and `sum` operators are very similar to the multiplication and addition operations. The major difference is that these reduction operations have a higher latency because they have to feed their own results back into themselves in order to perform the full reduction. From table 3.3 it is clear that the `prod` operation is seldom found in the Perfect Club programs while the `sum` operation appears in almost all of them, albeit in small proportions.

In the memory category, we have included vector loads and stores as well as the indirect memory accessing functions, `gather` and `scatter`. With respect to loads and stores, we see that overall there are 52% more loads than stores (using a weighted average). It has already been reported for scalar programs [CB94b] that it is more common to have load instructions than store instructions. Also in [VH92] a similar relation was found for code compiled on a Cray Y-MP vector multiprocessor. It is interesting to note that for all programs the percentage of store operations remains between a 10% and a 20%.

Program	Total ($\times 10^6$)	Compute %							Memory %			
		+/−	×	÷	$\surd$	$\prod$	$\sum$	oth	ld	st	gth	sct
swm256	9534	31	21	0.8	–	–	–	0.0	33	14	–	–
hydro2d	3974	15	14	2	0.1	–	0.5	24	33	11	–	0.0
arc2d	4087	20	29	2	0.4	–	0.1	2	29	14	4	–
flo52	1242	24	16	2	0.1	–	0.1	2	38	18	0.0	–
nasa7	3912	27	27	0.7	0.1	–	0.1	1	27	17	–	–
su2cor	3357	23	26	1.0	0.1	–	0.6	4	25	15	5	–
tomcatv	917	24	18	0.7	–	–	–	0.7	37	19	–	–
bdna	1590	25	22	1	1	–	1	6	26	18	0.2	–
trfd	1095	20	20	0.1	–	–	–	0.3	40	20	–	–
dyfesm	696	15	22	–	–	–	6	0.0	41	16	0.8	–

Table 3.3 Detailed breakdown of vector operations. Column Total is in millions of operations. Rest of columns are percentages over column Total (oth = other, gth = gather, sct = scatter).

3.4 VECTOR LENGTH DISTRIBUTIONS

A very important factor in achieving high performance in vector execution is the effective vector length used by vector instructions. Current vector architectures have very different sizes for vector registers, ranging from 64 elements in the Cray Y-MP, to 128 elements in the Cray C90 and Convex C3, and up to 8192 elements in the reconfigurable vector register file of Fujitsu's VP2400/10 machine. Generally, the higher the number of register elements used during the computation the better, because memory and functional unit latencies and vector start-up costs are amortized over the length of the vector register. Longer vector lengths have a lower associated overhead and, thus, achieve higher performance.

Our tools allow us to monitor all stores to the vector length register in the Convex C3 and thus we have an exact measure of all vector lengths used during the execution of a program. In Fig. 3.1 we plot the *cumulative* percentage that each vector length is actually used by a vector instruction. The X-axis plots the vector length value and the Y-axis plots the cumulative percentage of instructions that have used a certain vector length. For example, for program `nasa7` we can see that 60% of all vector instructions were executed with a vector length that was lower than or equal to 30. The remaining 40% of instructions executed using a vector length very close to the maximum, 128.

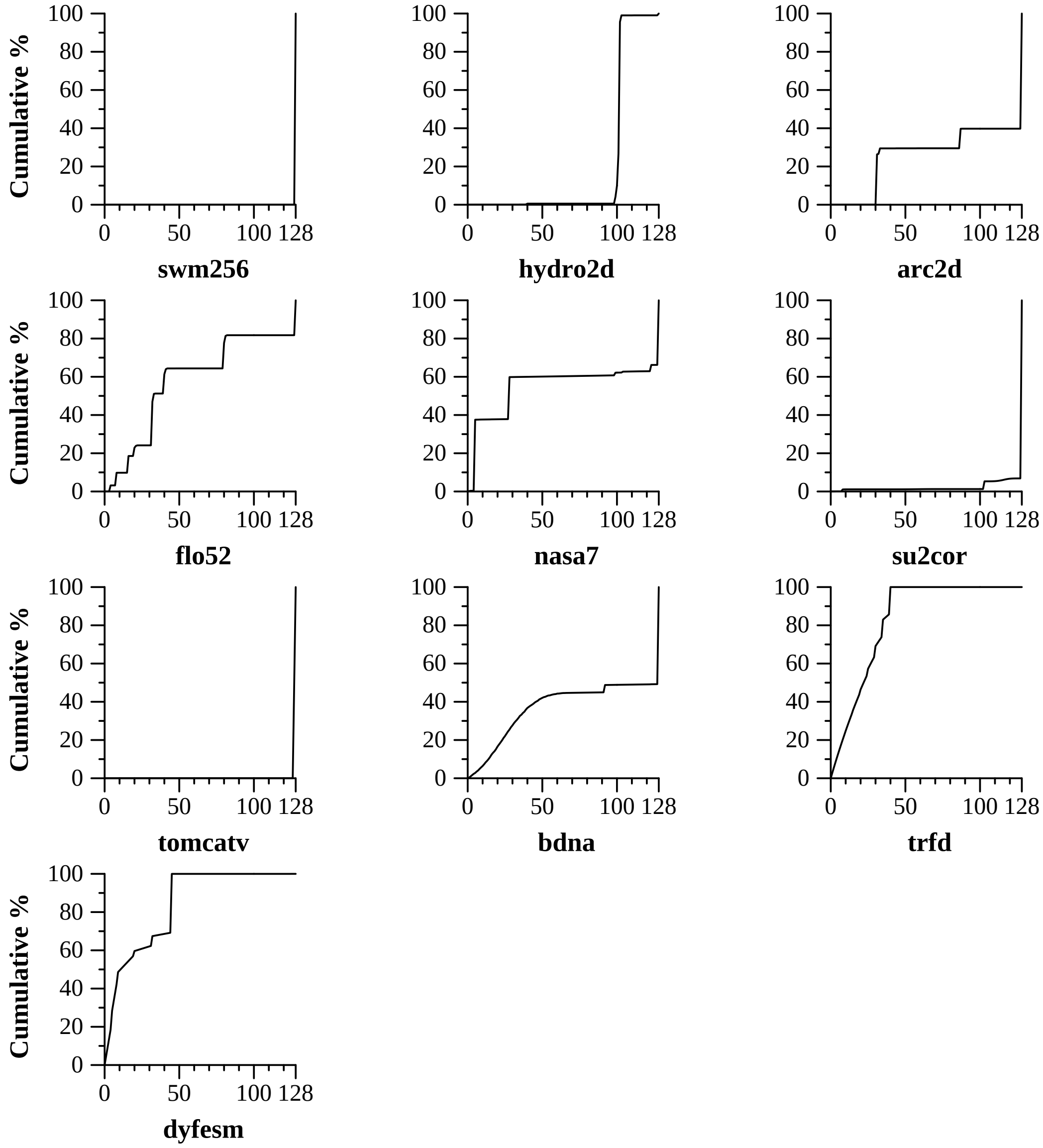

Figure 3.1 Distribution of vector lengths used. X-axis is the VL value and Y-axis is a cumulative percentage distribution.

From Fig. 3.1 what is clear is that the vector length distributions do not follow any regular pattern. Three programs behave as expected (`swm256`, `su2cor` and `tomcatv`), having the majority of their vector lengths clustered around 128 and having a small percentage (below 5%) of very short vector lengths that are the residuals generated due to strip-mining. `Hydro2d` has a single dominant vector length (102) that comes from the fact that this value is the number of grid points used in the z-direction of the problem. The other programs use many different vector lengths, with distributions either following a staircase model or a slope model. For example, `trfd` has a very regular slope that is the result of its main loops working on triangular matrices. Two programs, `dyfesm` and `trfd`, have almost all vector lengths shorter than 64 elements.

All in all, this data suggests that even among vectorizable programs the utilization of the vector registers of the C34 machine varies a lot. While some of the programs almost always use the 128 elements, the others could also run at a similar performance on machines having shorter vector registers. It is worth re-emphasizing that for the Perfect Club subset, the working sets have been artificially reduced and this might explain some of the smaller vector lengths observed.

3.5 VECTOR SPILL CODE

Many of today's vector architectures have relatively few vector registers – most likely because of the hardware cost of vector registers at the time the architectures were defined.

The Convex C34 and the CRAY Y-MP, for example, only have 8 vector registers. This limited number of registers makes the task of the compiler more difficult, because it has to balance aggressive scheduling techniques against a very limited number of vector registers. When compiling large dependence graphs, the register allocator usually runs out of available free registers and has to spill some live vector variables to the stack and later reload them. This spill code will be very negative in an architecture where there is only one memory bus. Every spill load/store instruction inserted by the compiler will result in a net increase of total memory traffic. Since we already saw in section 3.2 that the benchmark programs are memory bound, this net increase in memory traffic will most probably not be hidden by other computations and will result in a decrease in performance.

The compiler used to compile the Perfect Club codes puts all its spill code on the stack, and seldom uses the stack to hold vector data that it is not spill data. Thus, we have been able to distinguish spill memory operations from "normal" memory operations by checking if the memory access was to the stack or not. We have further enhanced our technique to detect spill code by creating a table (at runtime) of all data that is stored to the stack and checking that there is some load that later reloads it. If this is the case, then we annotate that both

Program	Total Traffic	Load Spill %	Store Spill %	Total Spill %	Avg. reuse
`swm256`	4498.5	7.0	7.0	14.0	1.0
`hydro2d`	1770.7	1.2	1.2	2.4	1.0
`arc2d`	1929.9	6.3	4.5	10.8	1.4
`flo52`	691.4	5.9	5.9	11.9	1.0
`nasa7`	1721.6	1.2	1.2	2.4	1.0
`su2cor`	1494.8	13.5	6.9	20.3	2.0
`tomcatv`	513.5	20.3	20.3	40.5	1.0
`bdna`	700.6	38.0	31.5	69.5	1.2
`trfd`	657.3	–	–	–	–
`dyfesm`	397.9	0.1	0.1	0.2	1.0

Table 3.4 Percentage of spill code in vector memory accesses. Column 2 is in millions of operations.

the store and the load were spill operations. If, by chance, there is a load or a store that references the stack but it is not in our table, this memory access is not considered to be spill code.

Using this technique, the total number of vector spills has been computed for each program. Table 3.4 presents the results. As it can be seen, there is a disparity in the amount of code found in each program, ranging from almost none in `dyfesm` and `trfd`, up to 70% in `bdna`. We have split the percentages of spill code belonging to loads and stores (columns 2 and 3) and also give the combined percentage (column 4). Column 1, total vector and scalar memory traffic is in millions of operations. Finally, we have also computed the average reuse of each vector spilled to memory. By dividing the amount of load spills by the amount of store spills, we get the average number of times that a vector spilled to the stack is later reloaded. The result is presented in column 5. For all programs the reuse is almost non-existent, that is, a vector is spilled and reloaded only once. Only in three cases, `arc2d`, `su2cor` and `bdna`, the degree of reuse is higher.

The percentage of spill for these vectorizable programs is rather remarkable. The case of `bdna` is clearly surprising and will be analyzed in the following section. All in all, the influence of having only eight registers in a single bus architecture is quite evident. All the spill code presented in table 3.4 will impact (negatively) the run time of the programs.

3.5.1 Example Cases of Spill Code

In this section we will present some examples of the typical spill code found in vector programs. As we mentioned in chapter 1, this thesis presents several methods to deal at run-time with spill code. This techniques will be introduced in chapters 5, 6 and 8. The code fragments that we present here are needed to understand the usual shapes of loops having spill code, and why the techniques we present do actually work. These examples will be later referenced in following chapters. The reader is free to skip this section at this point and later return to it without disrupting the natural sequence of topics being introduced.

We will look at the three programs having most spill code and present a detailed analysis of their main loops.

`Bdna`

Program `bdna`, with almost 70% of all its memory operations being spill code, has a very large single loop that contains more than 800 vector instructions. This loop makes heavy use of intrinsic calls, like `SIN` and `EXP`, that are translated into special library calls that are already vectorized. The loop turns into a large sequence of basic blocks, separated by calls to the intrinsic routines, that is full of spill code due to the extremely large number of live values and due to saving/restoring of vector registers at procedure call/return boundaries. Unfortunately, this large block is too big to be presented in a meaningful way here.

`Tomcatv`

Program `tomcatv` has around 40% of spill code, all concentrated in a single loop. This loop is compiled into a single basic block that contains 16 spill-loads and 16-spill stores, plus 18 "program" loads and 3 "program stores". Figure 3.2 presents a schematic view of the spill-loads, spill-stores and stores found in this basic block (normal loads have been removed for clarity). For each spill-load there is a back arrow connecting it to the store from where it is reading its data. We have labeled each arrow with the number of intervening stores between the spill-store saving a register and the spill-load reading it. For example, between spill-store 10 and spill-load number 10 there are a total of 4 stores (spills 10, 11, 12 and 13).

The intuition behind the number of intervening stores is that, if we are trying to dynamically eliminate spill code, we will need to hold all these intervening stores in some form of storage space internal to the processor so that the spill-load requesting that particular vector can be executed without actually going to main memory. This "storage space" inside the processor could be a cache or could be implemented using other mechanisms that will be described in chapters 5 and 6. We will avoid using the explicit term "cache" to prevent confusion and will defer further discussion of the nature of this storage space until later chapters.

By ordering the labels on each arc found in Fig. 3.2, we get the following sequence: 1, 2, 3, 3, 4, 4, 5, 6, 6, 7, 9, 11, 13, 14, 17, 18. From this ordering we can statically determine that, with a storage space being able to hold 4 vector registers, 6 spill-loads could be killed. Enlarging the storage space to 8, 16 and 32 vector registers would yield 10, 14 and 16 spill loads being dynamically eliminated.

In order to be able to dynamically eliminate spill-stores, the storage space would need to hold at least 20 vector registers, so that after the first iteration, we would have stores 1–19 in the storage space and when store #1 from the second iteration would try to enter the storage space, it would match against store #1 from the previous iteration and eliminate it.

`Su2cor`

For program `su2cor`, we investigated a total of 10 basic blocks that take around 80% of all the execution time. Among them, we found 4 that contained spill code. Each one of them takes 21.6%, 9.2%, 8.6% and 5.6% of all execution time. Figure 3.3 shows the schematic views for the two first basic blocks (labeled 466 and 478). The other two basic blocks have a very similar structure to basic block 466 so we omitted them for the sake of clarity.

Again, merging and reordering all the labels that appear in Fig. 3.3 we obtain the following sequence: 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 5, 6, 7, 8. The sequence shows that with a storage space of 4 vector registers we should be able to catch 8 spill-loads and enlarging the storage space up to 8 vector registers would catch all spill-loads for these two basic blocks.

3.6 PERFORMANCE LIMITS

The data presented so far is enough to establish several conclusions on the performance of our set of highly vectorizable codes. Our goal is to establish some bounds on the best possible

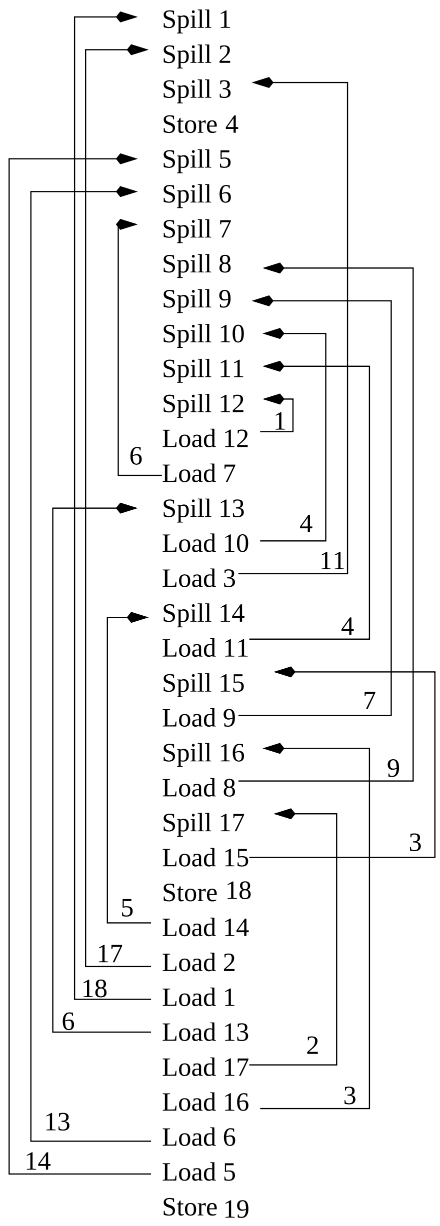

Figure 3.2 Schematic view of basic block #28 from `tomcatv`. Included are only spill-loads, spill-stores, and normal stores. All scalar code and vector arithmetic instructions have been removed for clarity. The label on each arc is the number of intervening stores between a spill-store and its matching spill load.

execution times for these programs. These bounds will be based on resource occupation and will help point out which resource is the bottleneck when executing our programs.

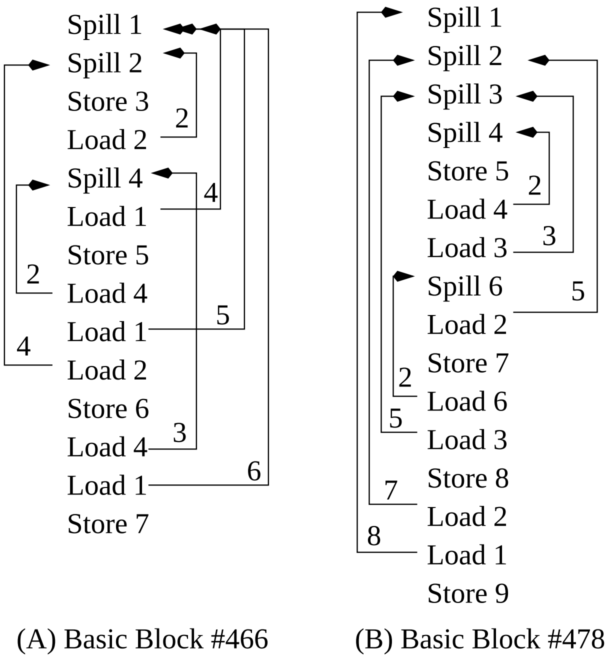

Figure 3.3 Schematic view of basic blocks #466 and #478 from `su2cor`. Included are only spill-loads, spill-stores, and normal stores. All scalar code and vector arithmetic instructions have been removed for clarity. The label on each arc is the number of intervening stores between a spill-store and its matching spill load.

Consider our reference architecture from a very high level point of view. It mainly consists of a fetch and decode unit, a memory port, two vector functional units and several scalar functional units. Looking back to tables 3.1 and 3.2, we can group all the data presented there in four main categories that reflect this broad division: memory accesses, vector computations, scalar computations and total instructions executed. Figure 3.4 presents in a graphical way the amount of operations found in each of these four categories.

What is the pressure on each of our major resources ? Whatever resource is the most used one, it will give us a lower bound on the best possible execution time for each program. For example, if a program has 1000 memory accesses, 200 vector computations, 50 scalar computations and has dispatched 150 instructions, we know it is impossible to execute it in less than 1000 cycles. At the very least, we will need 1000 cycles to dispatch all the memory requests. Thus the resource that is most used gives a lower bound on the minimal execution time of a program. Typically, the exact minimum execution time of a program will be larger than this bound that we are computing.

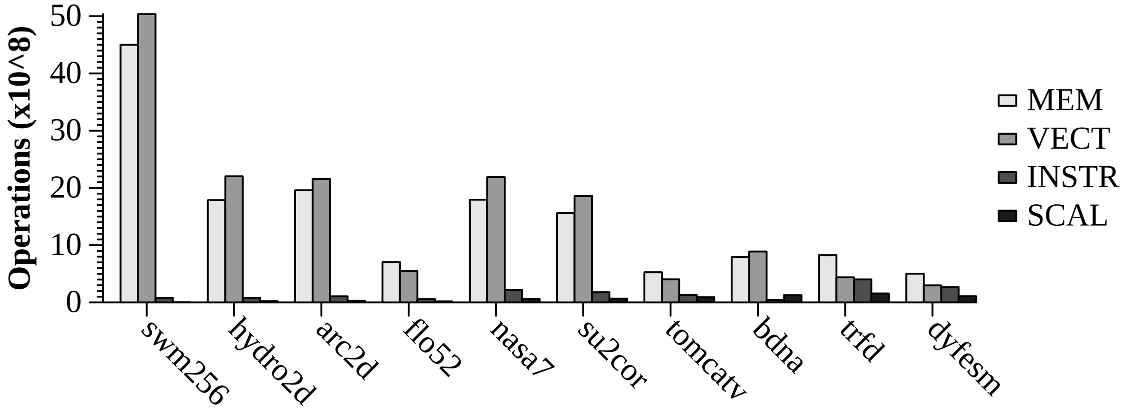

Figure 3.4 Comparison of memory accesses, vector computations, instructions decoded and scalar computations.

Looking at Fig. 3.4 it is clear that the fetch and decode unit and the scalar units have much less work to perform than the vector units and the memory port. The vector computations can be executed in two different functional units (although these two are slightly asymmetric). Thus the vector bar shown in Fig. 3.4 should be halved (taking into account the asymmetries, of course) to obtain the resource occupation of the two vector alu's.

Figure 3.5 presents a comparison of the memory accessing operations against the vector bar already halved as described (bar "VECT/2"). What this figure shows is that *all* programs are *memory bound* when run on our reference architecture. The memory port will be the most used resource during the execution of all these programs. This is a very important point since, as we will see later, the memory port will not be as busy as one would except looking at Fig. 3.5.

It is worth pointing out that these programs would probably be memory bound for most other architectures listed in table 2.1 (page 22) that have a memory to compute ratio of 0.5. Obviously, the measurements we are looking at mostly depend on the compiler generated code and on the fact that we only have 8 vector registers.

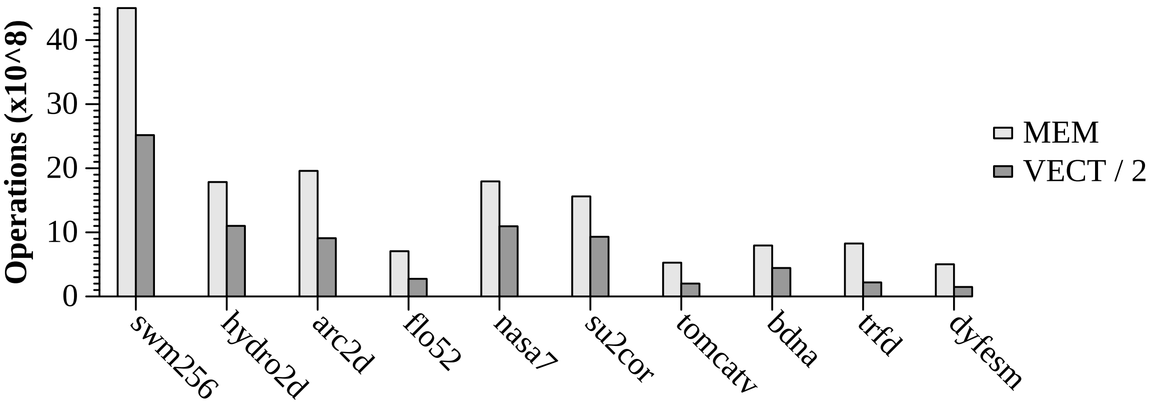

Figure 3.5 Lower bounds on execution time given by operations executed.

3.7 RELATED WORK

There have been relatively few detailed performance studies of vector machines. In [VH92, VSH91, Vaj91] the Cray Y-MP [Cra90] architecture is evaluated through a study of the scalar and vector instruction level characteristics. This work is based mostly on data collected by running a tool that gathers instruction execution statistics irrespective of time. Also, some time-dependent data, regarding instruction issue rates, were gathered with the Cray Y-MP Hardware Performance Monitor (HPM). The time-independent statistics are in some respects similar to those given here. However, because of the collection method used in [VSH91], the vector lengths were not known. That is, the number of vector *instructions* could be determined, but not the number of vector *operations*. Operation counts were estimated by using average vector lengths collected over the entire program for three major classes of instructions: floating point, integer, and memory. Our results do not have this deficiency – vector lengths for individual vector instructions are known so vector operation counts are much more accurate.

In [SH91, SH94] detailed measuring models of vector architectures are presented. Here, the emphasis is on analytical models, and some small kernel benchmarks are evaluated to establish parameter values for the models. Instruction level statistics are not given. In [Lee92] a study of the Cray Y-MP vector register usage was carried out and several alternative vector register configurations were proposed.

Malony et al. [MLR90] presented some low level analysis of three Perfect Club programs. Using a tracing library developed by themselves based on statistical sampling of the HPM counters of a Cray X-MP, they present data on number of instruction issued, number of memory references, etc. The key differences between their approach and the work being presented here is that while they use such data for application improvement, we focus on the bearing of the results on future vector architecture design. Moreover, our tracing method is more accurate than the HPM counters sampling and does not suffer from the perturbations they describe.

Williams et al. [WMK90], compare the performance of two different workloads running on an X-MP at two different sites during the weekend/night production periods. The present vectorization ratio as well as an approximation to the operation mix and some data on memory conflicts. As the authors indicate, they intended to provide some data that might be of use to future designers, but, nonetheless, focus a little bit too much on the differences between the two workloads rather than on future design issues.

The study carried out in this thesis is also important because represents the first with a machine having a compute to memory performance of 0.5. All these studies just cited have been made using the HPM counters on Cray machines, which have a 1+0.5 ratio (see table 2.1, page 22).

3.8 SUMMARY

In this study we have used a trace driven approach to gather data about the instruction level characteristics of vector programs. Our method has given us the full execution of the programs, including not only user code (like in many of the instruction level studies found in the literature) but also the library code with all the specific fortran math vector routines.

Using these traces we have presented the degree of vectorization in terms of operations of some Perfect Club and Specfp92 programs. We have seen how the proportion between compute and memory operations is almost 1 to 1. From the operation type distribution we have concluded that the two most important computation instructions are additions and multiplications (80% of all computation operations) and that they appear with almost the same frequency. Thus, designs that have a good balance between the number of units that perform additions and the number of units that perform multiplications will better match the behavior of programs.

For the vector specific measurements, we have looked at the vector length used in vector instructions. We have seen how the effective vector length used differs widely for each program and is very application dependent. For some codes, the average value could be rather low, indicating that an important fraction of the register space is being wasted and that the startup overhead of vector operations is not being compensated along a full vector.

Regarding the operation distribution of vector code, the balance between computation and memory accessing instructions has been measured. The results show that the programs are slightly unbalanced and have more computations. The key point presented in this chapter is that all benchmark programs are highly memory bound when run on our reference platform. Data presented shows that if vector computations are properly split between the two vector units then the memory port should become the main bottleneck during a program's execution. This data also gives us a bound on the maximum performance achievable by each program. Since memory accesses are the bottleneck of all ten programs, we know that they can not be executed in less cycles than the number of memory accesses they perform. The goal of the techniques presented in the following chapters will be to try to see if such a performance limit can be reached or not.

An intuitive conclusion that one might extract from the data presented is that the memory port will be busy during almost all cycles of the execution of a program. As the next chapter will show, this intuition is wrong and is actually the source of the inability to tolerate large memory latencies of traditional vector processors.

Regarding spill code, we have seen that the 8 vector registers induce a lot of vector spill code. The percentages vary a lot across programs, but 8 out of our 10 benchmarks did have spill code that amounted between a 10 and 70% of all memory traffic. We have examined the amount of vector spill code and have found that for the vectorizable set of programs it was quite high, representing over 10% of all memory operations in several cases.

4

INSTRUCTION LEVEL PARALLELISM

Summary

This chapter analyzes the performance of the benchmark programs when executed on the reference platform. We will look into three key issues: the effects of memory latency on execution time, the available instruction level parallelism and the occupation of the memory port. We will see that traditional vector machines have many deficiencies and that they are not very latency tolerant and do not fully use the full port bandwidth. We present examples and quantitative data of the hazards that prevent achieving maximum performance. We also look into the execution costs of vector spill code.

4.1 INTRODUCTION

This chapter analyzes the performance of the benchmark programs when executed on the reference platform. Our first goal is to determine whether they run at their maximum possible speed or not. Using the lower bounds presented in section 3.6 we will compare the performance achieved against the minimum ideal execution time. As the following sections will point out, it turns out that even the highly vectorizable programs (`swm256`, for example) do not achieve the peak MFLOP/s of the machine.

The second goal of this chapter is to examine program performance when memory latency is varied. We will study a range of latencies going from a perfect memory system having only 1 cycle latency up to a slow memory system having 100 cycles of latency. As we will see, execution time degradates almost linearly when memory latency increases. This observation will be the starting point for the dynamic instruction issue proposals introduced in chapters 5, 6 and 7.

The third goal will be to determine the reasons that prevent a highly vectorizable code to run at maximum speed on a vector machine. We will start by analyzing the instruction level parallelism available during the execution of each code. This analysis will reveal the fraction of time that we have the two vector units busy and will show that, unfortunately, most of the time, the machine is idle waiting for data to arrive from memory. Despite the fact that the occupation of the memory pipeline is critical to performance, a traditional, in-order, vector architecture is not able to keep the memory pipeline busy: the memory port remains idle for many cycles.

The fourth and last goal of this chapter will be to evaluate the execution time impact of spill code, using a very simple technique. We will artificially eliminate all loads and stores known to be spill code. Then we will compare the execution time of a program with and without spill code. The result will be an upper bound on the benefits that we will achieve when we put into work the dynamic load/store elimination techniques that we will propose in later sections.

4.2 EFFECTS OF MEMORY LATENCY

Our first goal is to compare the execution time of each program with its theoretical ideal execution time. Recall from section 3.6, that all our codes are memory bound and, thus, their minimum execution time is determined by the number of memory accesses they perform. This minimum value is an absolute lower bound on execution time.

Using the simulator for the reference architecture with the parameters described in section 2.4, we execute the ten programs with several memory latencies. Here the term *latency* refers to the number of cycles elapsed from the point where we request a word from the memory system to the point where the word arrives to the processor. Recall that we do not model conflicts in the memory system and we assume that it always delivers one datum per cycle. Also, stores do not have latency, since their latency is only seen by the final memory modules.

We start with a latency of 1 cycle, which represents a perfect memory system that will yield the best possible performance. Then we increment latency up to a 100 cycles, a value significantly larger than the latencies currently found in parallel vector processors.

To put in perspective this large range of memory latencies, consider current vector processor cycle times, in the 2ns–4ns range, and current memory technology: DRAMS, the cheapest form of memory, have a cycle time of 100ns [Pri96, page 96]; Synchronous DRAMS (SDRAMS) improve the cycle time and chips have been announced with cycle times as fast as 8ns [Pri96, page 166]; finally, SRAMS, the fastest and most expensive form of storage can have cycle times in the range of 5–10ns [Pri96]. These three different technologies have very different density characteristics as well as different effective data rates: SRAMS can truly sustain one datum per cycle, while DRAMS and SDRAMS require periodic refreshes and use at least two cycles (RAS and CAS) to send an address to the memory module.

If a memory system was to be implemented in DRAM technology, we would have a module latency of over 50 cycles approximately. Crossing the several stages of a multi-stage crossbar would cost some extra cycles of round-trip latency. Therefore we believe that although cheap DRAM vector memory systems are realizable, they have a very large latency associated with them. Since we aim at producing high performance yet cost-effective vector architectures, we want to be able to deal with latencies in the 100 cycles range.

As a point of comparison, Smith, Hsu and Hsiung [SHH90] present the evolution of memory latency in several vector supercomputers, both in absolute and relative terms. In relative terms, that is, expressed in processor clock cycles, latency has steadily degraded: the X-MP and Y-MP lineage have evolved from around 10 cycles up to 20–25 processor cycles. The NEC SX-2 and SX-3 had, respectively, a latency of 35 and 65 processor cycles. Recently, a version of the NEC SX-4, the SX-4/A has been built with a large capacity SDRAM main memory, but no data on the latency of such machine is yet available.

By studying a perfect memory system with 1 cycle latency, we will show that even in such ideal conditions, in-order vector architectures have architectural problems that prevent them from achieving maximum performance. By increasing the latency up to a 100 cycles we want to show that traditional vector architectures are not truly latency tolerant and that they require some additional techniques, such as the ones proposed in chapters 5–7, to be able to effectively run under large memory latencies.

Figure 4.1 presents the execution time for the ten benchmarks programs when memory latency is varied from 1 up to 100 cycles. In each graph we also show the minimum absolute execution time that can theoretically be achieved (curve “IDEAL”, along the bottom of each graph). To compute the IDEAL execution time for a program we use the total number of cycles consumed by the most heavily used vector unit (FU1, FU2, or MEM), as already discussed in section 3.6. Thus, in IDEAL we essentially eliminate all data and memory dependences from the program, and consider performance limited only by the most saturated resource across the entire execution.

The graph shows two very important points. First, vector architectures can not tolerate large increases in memory latency. In all programs, the degradation in execution time is almost linear when memory latency increases. Second, even with a perfect memory system, that is, with a latency of 1 cycle, the distance between the simulated execution time and the IDEAL line are very significant. Thus, even in the absence of memory latency, there is some factor that limits performance of our reference vector machine.

Figure 4.2 presents the same data as Fig. 4.1 but in a relative way. Each axis in Fig. 4.1 has a different scale and thus it is difficult to gauge the real magnitude of the memory latency effects. Figure 4.2 presents for all programs the slowdown with respect to their IDEAL execution time when memory latency is increased. The “IDEAL” line for each program is considered to be the base performance value (1.0) and the simulated execution times, at latencies 1–100, are plotted relative to this base value.

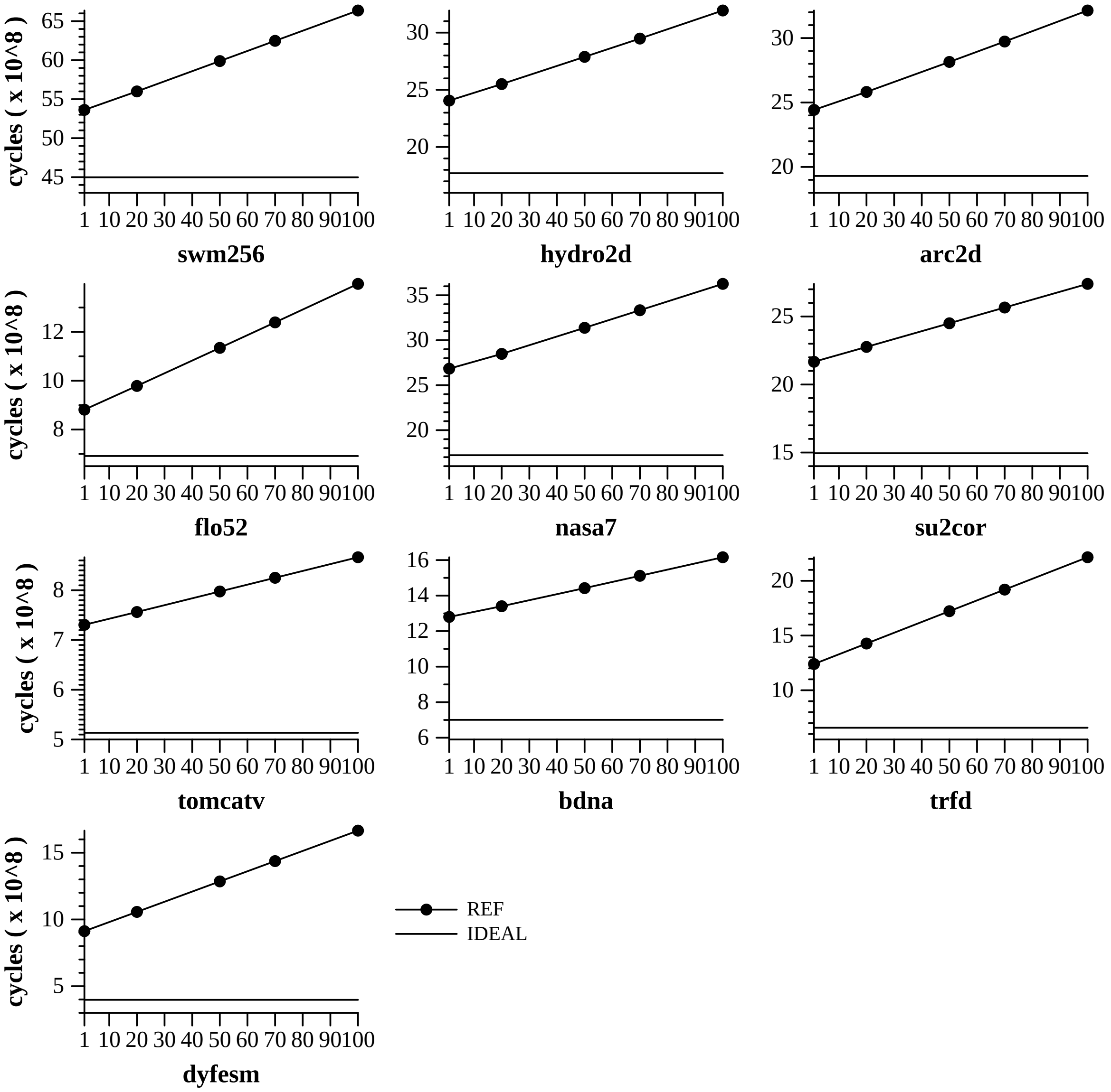

Figure 4.1 Execution time versus memory latency. The "IDEAL" line represents the minimum execution time determined by the most saturated resource, the memory port. X-axis is memory latency expressed in processor cycles.

We can see from Fig. 4.2 that performance for most programs is far away from the optimal execution speed and that the degradation due to memory latency can be as high as a 80%. Even with a perfect memory system programs are between a 20% and a 60% away from their peak speed. This implies that some architectural improvements should be included so

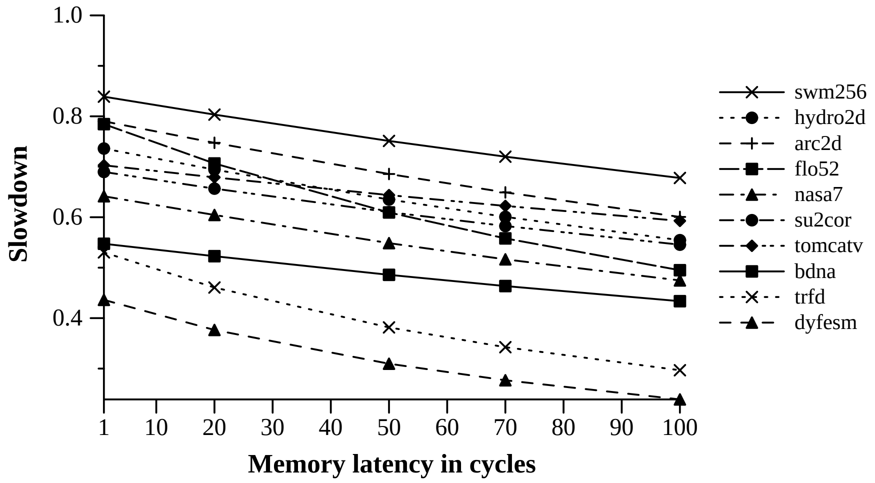

Figure 4.2 Slowdown with respect to the IDEAL execution time for each program when memory latency is varied. The IDEAL execution time of each program (refer to Fig. 4.1) is set at 1.0.

that, at least when the memory system is perfect, programs could run closer to their peak performance.

4.3 INSTRUCTION LEVEL PARALLELISM

Why do programs degrade so much when memory latency is increased ? To answer this question we need to investigate the instruction level parallelism (ILP) realized when our benchmark codes are run on the reference machine. The achieved ILP gives us an idea of how the computations are distributed over time and at what fraction of a full program run we are achieving the peak speed.

Consider only the three vector functional units of our reference architecture (`FU2`, `FU1` and `MEM`). At any given point in time, the machine state can be represented with a 3-tuple that represents the individual state of each one of these three units. For example, the 3-tuple $\langle FU2, FU1, LD\rangle$ represents a state where all units are working, while $\langle\ ,\ ,\ \rangle$ represents a state where all vector units are idle. The execution time of a program can thus be split into eight possible states.

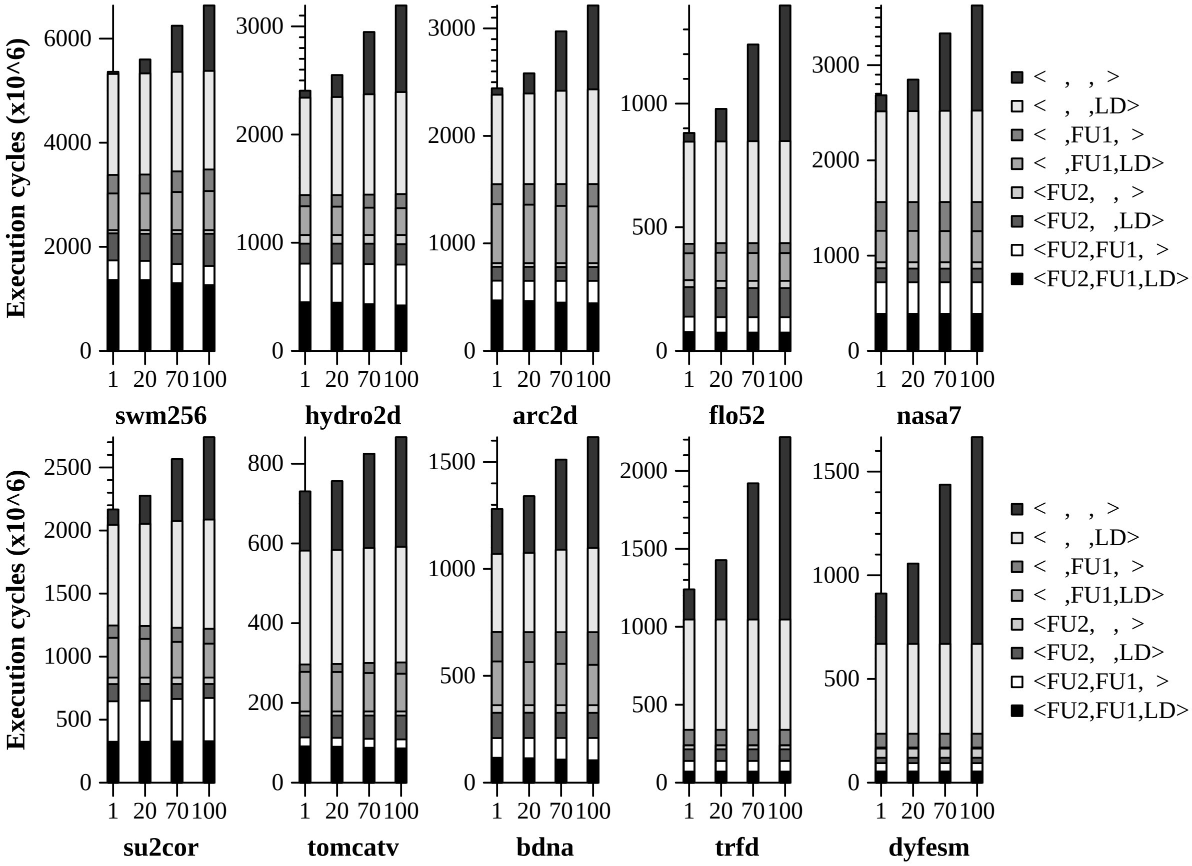

Figure 4.3 Functional unit usage for the reference architecture. Each bar represents the total execution time of a program for a given latency.

By analyzing these 8 states across several memory latencies, we can understand the impact of changing the memory performance on the behavior of the programs. Figure 4.3 presents the splitting of the execution time into states for the ten benchmark programs. We have plotted the time spent in each state for memory latencies of 1, 20, 70, and 100 cycles.

From this figure we can see that the number of cycles where the programs proceed at peak floating point speed (states $\langle FU2, FU1, LD\rangle$ and $\langle FU2, FU1, \rangle$) is quite low. The number of cycles in these states changes relatively little as the memory latency increases, so the fraction of fully used cycles decreases.

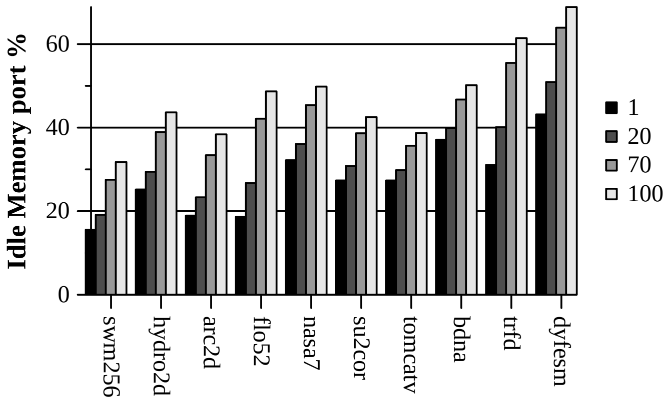

Figure 4.4 Percentage of cycles where the memory port was idle, for 4 different memory latencies.

Note that when memory latency is increased, the processor spends more cycles completely idle, in state $\langle\ ,\ ,\ \rangle$, waiting for data to arrive from main memory. The impact of this latency increase is higher for those programs that have relatively small vector lengths (`dyfesm`, `trfd` and `flo52`), but is otherwise present in all of them.

The most striking point from Fig. 4.3 is the large number of cycles where the memory port is idle (cycles where the processor is not issuing memory requests). If we add together the four states where the memory port is idle ($\langle\ ,\ ,\ \rangle$, $\langle\ , FU1,\ \rangle$, $\langle FU2,\ ,\ \rangle$ and $\langle FU2, FU1,\ \rangle$), we get a very high fraction of port underutilization. To highlight the importance of these cycles, Fig. 4.4 presents the percentage of port idle time over total execution time. At latency 70, the port idle time ranges between 30% and 65% of total execution time, but even with a perfect memory system the idle time is typically between 20% and 30%.

Combining the results from Fig. 4.3 and 4.4 and the data presented in table 3.2, we can see that these ten programs are clearly memory-bound and yet, they can have the memory port idle for up to a 50% of its total execution cycles. This is a counter-intuitive result, since the fact that the programs are memory bound would lead us to conclude that its memory port would be the most saturated resource and that it would be busy for the majority of cycles. The port idle time is responsible for the lack of latency tolerance of our reference architecture. In order to be latency tolerant, an architecture must issue a continuous flow of requests to the memory system, so that the latency of each individual request is masked by overlapping other requests. What happens in our reference machine is that, due to reasons we will detail in section 4.4, the processor is not able to make more than 1 concurrent request.

In order to extract the maximum performance from the architecture, this port idle time should be reduced to a minimum. Introducing some kind of "out-of-order" execution techniques we should be able to dispatch vector memory operations whenever the memory port is idle. While vector architectures are able to hide part of the latency by using the vector principle, there is still much room for improvement.

4.4 LIMITATIONS TO ILP

We have already identified the major source for the lack of latency tolerance of our reference vector architecture: there is very high fraction of cycles where the vector unit is completely idle simply waiting for data to arrive from memory. Our next step is to correlate this fact with common instruction sequences. That is, we want to quantify the situations that lead to all functional units being idle. Moreover, we would also like to know more about the behavior of whole loops. By understanding how full inner loops interact with main memory latency we will be able to motivate the introduction of dynamic instruction scheduling techniques.

We will start by giving some motivating examples of the behavior of loops and we will micro-analyze them in detail to highlight the hazards that occur during their execution. Once these types of hazards have been identified, we will give quantitative data showing their relative importance.

4.4.1 Compute-bound and Memory-bound loops

How can the machine become completely idle, simply waiting for data to arrive from memory ? If the programs are memory bound, how is it possible that at some points we do not have memory instructions ready to be issued ?

Since reasoning about full programs is rather hard, we will need to move to a greater level of detail and analyze vector loops. It is widely known that scientific programs spend most of their time in loops. Thus understanding the behavior of simple loops helps in reasoning about full program performance.

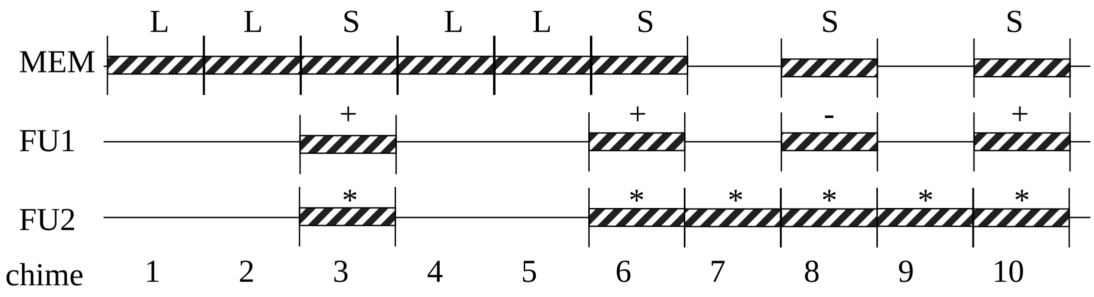

Figure 4.5 Execution of a memory-bound loop under a perfect memory system.

We can distinguish vector loops in three main categories: memory bound loops, compute bound loops and balanced loops. Let M be the minimum number of cycles required to execute all vector memory operations of a loop. Let C be the minimum number of cycles required to execute all vector computations of a loop. A memory bound loop is a loop where $M > C$, a compute bound loop is a loop where $C > M$ and, finally, a balanced loop has $M = C$. Roughly speaking, on our reference machine, a compute bound loop will have more than twice the computational instructions than memory instructions.

The important part of this classification is that we would expect that the performance of a memory bound loop is determined only by its memory instructions. Similarly, the performance of a compute bound loop should be determined only by its computation instructions.

Let's look at two loops extracted from our benchmarks. Figure 4.5 presents a memory bound loop. In the figure, time flows along the X-axis and has been divided in *chimes*. A chime is the time taken to execute a vector instruction with a vector length of vl. The peak performance of this loop is 8 chimes per iteration. The 10 vector computations could be executed in 6 chimes (recall that only FU2 can do multiplications) and the 8 memory operations could be executed in 8 chimes. Therefore, peak speed is determined by memory operations. Nonetheless, looking at Fig. 4.5, we see that there are 2 empty chimes in the memory stream. These have been caused by problems in the computation stream. For example, the empty cycles at chime 7 are due to the conflicts between the three multiply instructions found in chimes 6, 7 and 8. The store in chime 8 is dependent on the multiplication of chime 8, which, in turn, could not be executed before the two other multiplications.

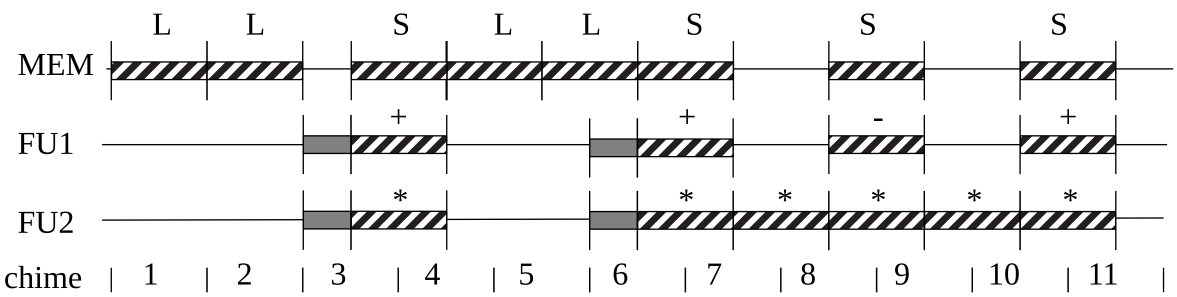

Figure 4.6 Execution of a memory-bound loop under real memory latency. Memory latency is represented by the grey bars on the FU1 and FU2 lines, at the start of chimes 3 and 6.

The major lesson learned from this loop is that the in-order nature of the reference machine forces the control unit to stall at chime 7, even though the memory port is idle. Since this is a loop, we could use the two empty memory chimes to start the first two loads belonging to the next iteration. If that was possible, then we would be running at peak speed.

What if we consider real memory latency ? Figure 4.6 shows the same loop again but adding a certain amount of memory latency in the instructions found at chimes 3 and 6. These four instructions use the two vector registers loaded by the load instructions of chimes 1, 2, 4 and 5. This second example shows that the in-order machine has its execution time further degraded when memory latency is introduced. From the theoretical 8 chimes, the loop took 10 chimes on a perfect memory system and takes 10 chimes plus two periods of memory latency to complete one iteration. On the other hand, if dynamic scheduling could execute the loads from the second iteration in the empty memory chimes, then these two periods of memory latency would be hidden by other load and computation instructions.

Let's look now at a compute bound loop. The loop in Fig. 4.7 has 28 computational instructions that could be executed, theoretically, in 14 chimes. It has 12 memory instructions that could be executed in 12 chimes. Yet, the loop takes 20 chimes to complete a full iteration. The reason now is that the memory part of the loop is interfering with the computation part. Since several loads must be done before starting the bulk of arithmetic operations, the in-order execution model can not successfully overlap memory accesses and computation.

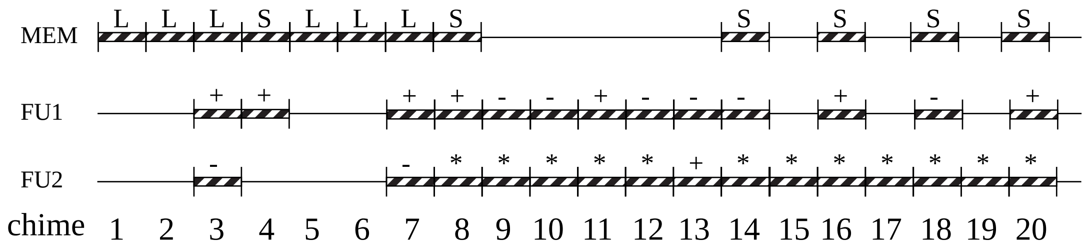

Figure 4.7 Execution of a compute-bound loop under a perfect memory system.

Later in the loop, when the main part of the computation is running, we have 5 empty chimes that should be used to start fetching in advance the loads for the next iteration. Towards the end of the loop, we have a similar example of empty chimes in the memory stream due to contention for the multiplier.

All in all, these two different examples show that when a simple in-order execution model is used, the memory accessing stream and the computation stream interfere with each other and prevent achieving peak speed. These examples show that using dynamic instruction scheduling, the memory accessing instructions of iteration $i+1$ of a loop could be started in the empty chimes of iteration i. The dynamic scheduling can be either a decoupling technique, where the two streams, computation and memory accesses, are split and sent to independent control units, or an out-of-order plus register renaming scheme that reorders and mixes instructions from several iterations of a loop.

4.4.2 Hazards

We will now quantify the different hazards occurring in the vector part of the reference architecture. We are particularly interested in determining which sequences of code force the machine to go into the idle states. Of especial interest is the $\langle\ ,\ ,\ \rangle$ state, but all states that cause the memory port to be idle are important.

There are many code sequences that lead to these problematic states. Figure 4.8 shows examples of the five major sources of stalls in our reference machine. These hazards are described as follows:

(a) LD hazard : whenever the code has two consecutive memory instructions, the second instruction will have to wait for the first to release the memory port. At that point, the issue unit will stall until the port is free.

(b) LDX hazard : when an instruction requires the data being loaded by a previous memory instruction, it will have to stall until all data items have arrived to the processor. Recall that there is no chaining between a memory load and a computation instruction. Thus, in the example shown in Fig. 4.8 (b), if we assume a $vl = 100$ and a memory latency of 50 cycles, the `add` instruction will have to wait $100 + 50$ cycles to begin execution. Of those 150 cycles the memory port will be used only for 100 cycles (the machine being in state $\langle\ ,\ , LD\rangle$), while the other 50 cycles will correspond to state $\langle\ ,\ ,\ \rangle$. This type of hazard is the most clear case where the lack of latency tolerance of the in-order machine can be seen.

(c) FU2 hazard : two consecutive instructions of type `mul`, `div` or `sqrt` will contend for the same functional unit, the FU2 unit, and thus the second instruction will stall unit until the first one completes.

(d) FU1 hazard : When an add-like instructions attempts to issue and the two functional units are already busy, the add instruction will have to stall until a unit is freed. This hazard can prevent later memory instructions from executing and using the memory bus.

(e) PRT hazard : This hazards involves a register file port conflict between two instructions. Recall that registers are grouped in pairs, `v0` with `v4`, `v1` with `v5`, and so on, with each pair having one write port and two read ports. In the example in Fig. 4.8, registers `v1` and `v5` are on the same bank and, therefore, it is not possible to have two simultaneous writes into the same bank.

Although in the five examples given we only highlight a single cause of stall, it is clear that these five hazards can appear in combinations. That is, an instruction might have to block because it lacks a functional unit (FU2 hazard, for example), because of a port conflict (PRT hazard) and because its input data has not yet arrived from memory (LDX hazard).

Table 4.1 presents a breakdown of the causes of stall for the reference architecture. Each row in the table corresponds to a combination of hazards that make the decoder stall. The first five columns correspond to the hazards just described and shown in Fig. 4.8. A value '1' in one of the first five columns in table 4.1 indicates that the machine stalled because of that particular type of hazard. For example, row 10 represents the situation were the decoder had an add-like instruction and could not issue it because of three different reasons: first, the two functional units were busy. Second, one of the operands of the instructions was being

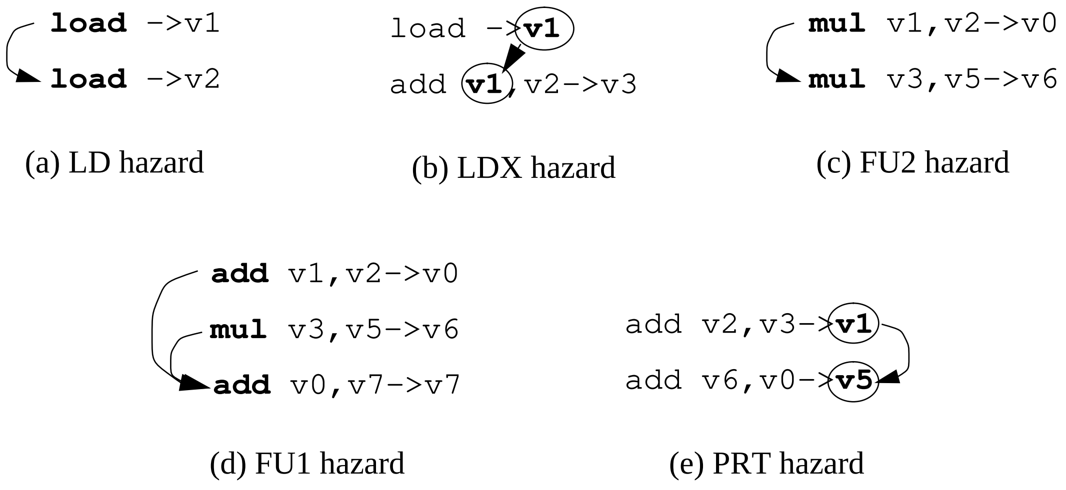

Figure 4.8 Examples of the five major hazard types in the reference architecture.

brought from memory by a load instruction and, thus, the instruction could not be chained to it. And third, there was a port conflict between one of the running instructions and the instruction about to issue. The last two columns show the percentage that each one of these hazard combinations represent over the total number of executed cycles (using the weighted and arithmetics means).

In table 4.1, we can see that the two most important hazards are the memory port hazard (column LD) and the load chain hazard (column LDX). The LD hazard is responsible on its own for 25.1% of stalls and adding up all rows where we have a 1 in column LD this percentage goes up to 28.5%. The LDX hazard is directly responsible for stalling the machine during 15.3% of all executed cycles and participates in another 7.3% of lost cycles, for a total of 22.6%. The last four rows in the table indicate that there is a relatively high pressure on the FU2 unit, since we had to stall the machine in 12.8% of all cycles waiting for this unit to be free.

Detailed Row Analysis

Row 0 presents the percentage of cycles that the decoder stalled due to the inability to chain a computation to a load. Having a 1 only in column `LDX` means that we were ready to issue a computation instruction, we had a free functional unit, we had ports to access the vector

	Hazard combination					% of execution cycles	
Row	FU2	FU1	LD	PRT	LDX	weighted mean	arithmetic mean
0.	0	0	0	0	1	15.3	17.0
1.	0	0	0	1	0	1.4	1.0
2.	0	0	0	1	1	1.9	2.0
3.	0	0	1	0	0	25.1	26.0
4.	0	0	1	0	1	0.9	0.0
5.	0	0	1	1	0	2.5	2.0
6.	0	0	1	1	1	0.0	0.0
7.	0	1	0	0	0	3.0	2.0
8.	0	1	0	0	1	1.0	0.0
9.	0	1	0	1	0	1.8	1.0
10.	0	1	0	1	1	0.3	0.0
11.	1	0	0	0	0	6.6	5.0
12.	1	0	0	0	1	2.0	1.0
13.	1	0	0	1	0	3.0	2.0
14.	1	0	0	1	1	1.2	0.0

Table 4.1 Distribution of major hazards in the reference architecture.

registers needed as operands, but one of those operands was the result of a previous load still in progress. The percentage of cycles lost in this situation is very high (15.3%). If we add together all rows where `LDX` is 1, we see that the lack of chaining with loads is co-responsible for as much as 23% of stall cycles. Other vector machines, like the Cray-YMP do not have this chaining restriction and can better utilize the functional units. Alternatively, if the machine had more registers, these cycles could probably be filled with useful computations by unrolling the loops and scheduling the operations such that vector loads and dependent computations could be moved apart as much as possible.

Row 1 presents the number of lost cycles due exclusively to port contention. Even though this row does not represent a very large fraction of all executed cycles, if we add together all rows where column `PRT` has a 1, we see that there is a high number of wasted cycles (12.1%) due to port contention. The architecture provides 8 read ports and 4 write ports to the vector registers and the maximum possible simultaneous requests are 4 reads and 2 writes for the functional units and one additional read or write for the `LD/ST` unit, but still does not seem enough to sustain the bandwidth required by the vector functional units. We have looked in detail at this port conflicts and found that the overwhelming majority of conflicts are due to reads.

Row 3 is the one that accounts for the largest percentage of stall cycles (25.1%). It represents the hazards that occur when the decoder tries to issue two consecutive vector memory instructions. This is a very common situation in vector loops for several reasons. If the compiler does not perform software pipelining and/or loop unrolling, the typical starting code of a loop is a sequence of loads that will bring the data to be operated on. This sequence of loads will always conflict in our architecture. Also, at the end of the loop there is usually a sequence of several stores that will conflict between them and will also conflict with the load instructions at the beginning of the next iteration of the loop. If we add all rows where `LD` is 1, we can conclude that the lack of more memory units is co-responsible for stalling the machine in 26% of all executed cycles.

Row 4 presents a very special case. Whenever we have a load followed by a dependent store, we have two conflicts. First, we have a 1 in column LD because we need the load/store unit to issue the store but it's busy servicing the load. Second, even if we had a second memory unit, the load chain restriction would prevent us from issuing the store. Thus, we also have a 1 in column LDX. Note that this situation is fairly rare, and corresponds mostly to loops that do initialization work, like copying arrays or initializing memory. Row 6 correspond to a still more special case in which the store following the load could also not be issued due to the fact that the read ports of the corresponding register bank were both busy (most probably, some other previous instruction is using them).

Row 5 is a common situation. Consider the following code:

```
1. add v2,v3,v0
2. st  v0,effa
3. ld  effa2,v0
```

This code can easily arise in loops. The result of the addition is stored in some array but is no longer needed in the rest of the loop body. After instruction 2, the compiler knows that register `v0` is dead, and reuses it to bring some other value from memory. When trying to issue instruction number 3, we have a `LD/ST` functional unit conflict and also a write port conflict to register `v0`, because the only write port to the register bank where `v0` belongs is busy servicing instruction number 1.

Let's consider rows 7 through 10. When we have a 1 in column `FU1`, its meaning is that *both* functional units were busy and so we could not issue a FU1-type instruction. These four rows indicate that the programs had some more parallelism available that could be exploited by adding more functional units. If we ignore for the moment rows 8, 9 and 10 which present port/load chain conflicts, the lost cycles that could have been successfully used with a third functional unit is *at least* a 6.1% of all executed cycles. Actually, if a third functional unit was added we could probably reduce more cycles of execution.

As we have also noted in the previous sections, the pressure on the `FU2` unit is very high due to the fact that `FU1` is not able to execute multiplications, divisions and square roots. This can easily be seen in rows 11 to 14, where the need for another functional unit of type `FU2` stalls the machine in 12.8% of all executed cycles. It is also true that rows 12 and 14 also have the load chain conflict but they represent only a 2.2% of the executed cycles.

4.5 TIME EVOLUTION OF ILP

The two previous sections have presented the average utilization of the functional units and have discussed several reasons why the programs do not achieve the peak floating point performance of the machine. These two sections have presented global averages for the behavior of the benchmark programs. Nonetheless, it would be very misleading to believe that programs behave in a very uniform way. On the contrary, and as has already been presented in [RR94], programs do not behave uniformly across all their execution time, and they present hot spots with more memory activity and hot spots with more cpu activity. In this section we will look at the evolution in time of the ILP available in our codes.

To measure the behavior of the benchmarks selected we have used JINKS to partition the full execution time of a program into "windows" of 10^5 cycles. During the cycles of each window, JINKS computes the average utilization of the memory bus and the combined utilization of the two functional units. This average utilization is simply the total number of cycles where a given unit is busy divided by the length of the window (10^5 cycles). For the memory bus the average utilization is a number between 0 and 1 and for the combined functional units the utilization is a value between 0 and 2. Then we plot all the average utilizations computed by

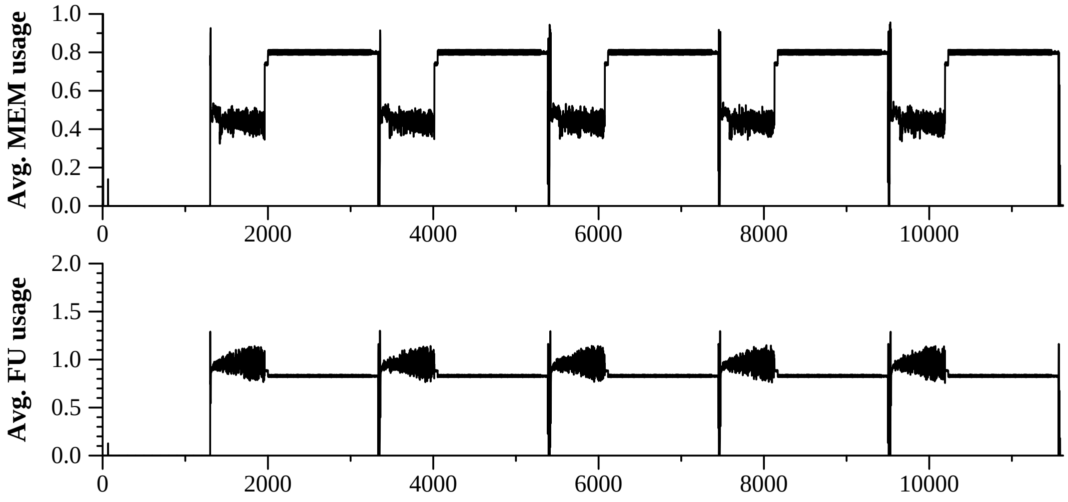

Figure 4.9 Time evolution of instruction level parallelism for `bdna`. X-axis represents time in window units. Each window is the average of 10^5 cycles.

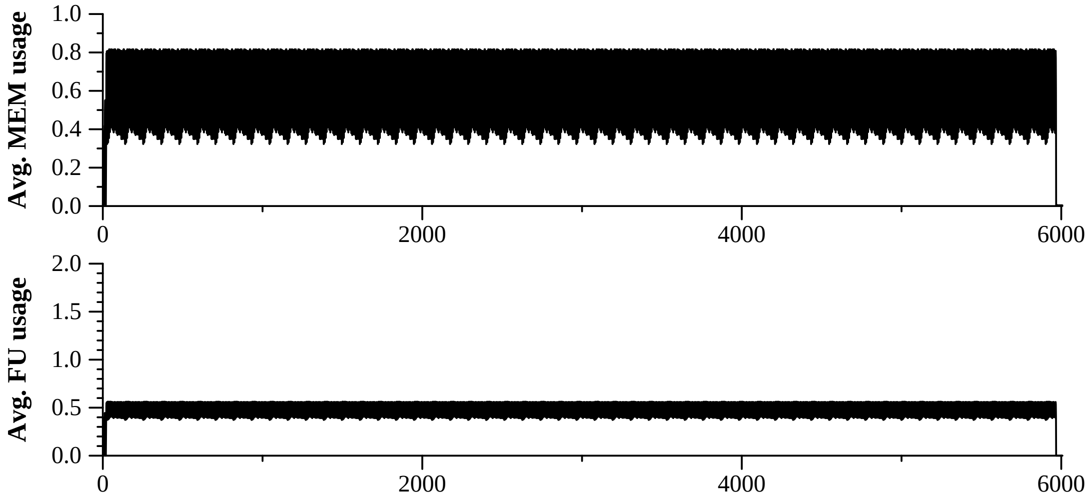

Figure 4.10 Time evolution of instruction level parallelism for `dyfesm`. X-axis represents time in window units. Each window is the average of 10^5 cycles.

JINKS in a sequential graph to get the evolution of the average utilization during the entire program execution.

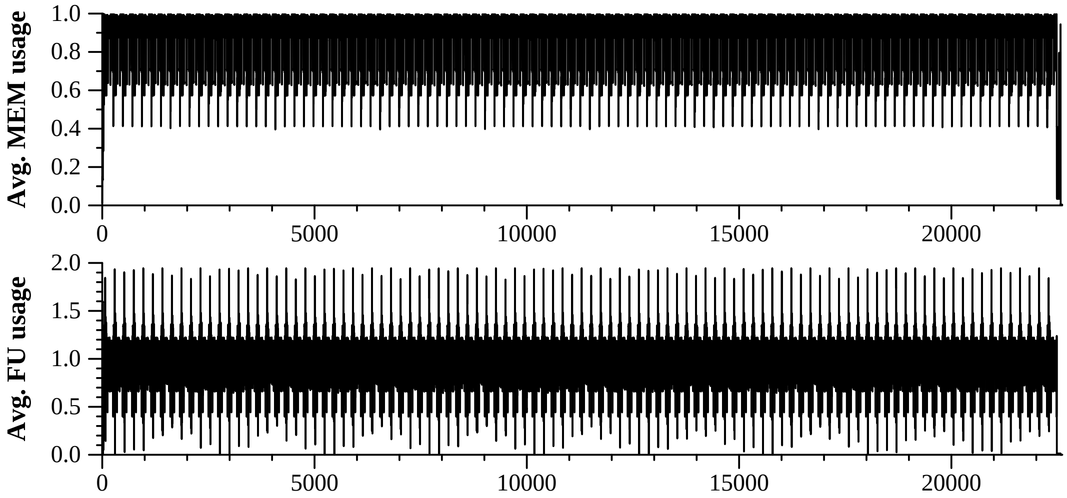

Figure 4.11 Time evolution of instruction level parallelism for `arc2d`. X-axis represents time in window units. Each window is the average of 10^5 cycles.

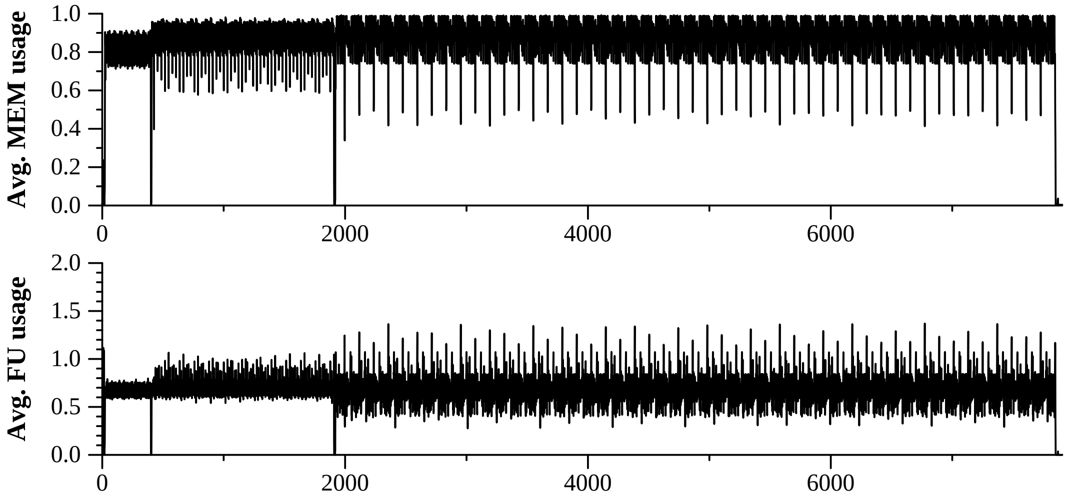

Figure 4.12 Time evolution of instruction level parallelism for `flo52`. X-axis represents time in window units. Each window is the average of 10^5 cycles.

Figures 4.9–4.13 plot the ILP evolution for six of our benchmarks programs. The others have been omitted for brevity. In each figure the top graph plots the evolution of the average memory usage and the bottom graph plots the combined utilization of both arithmetic functional units. Both graphs have exactly the same values in the X-axis. This means that the two

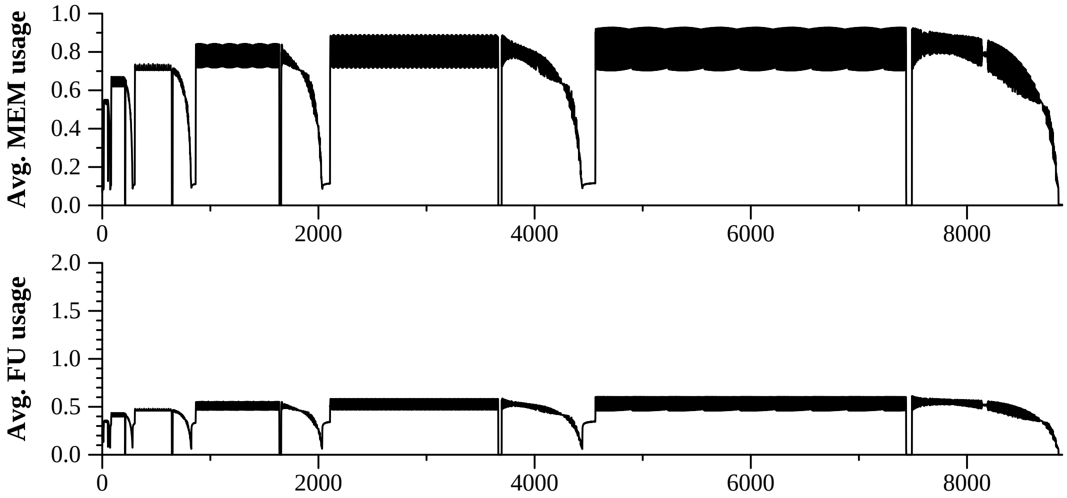

Figure 4.13 Time evolution of instruction level parallelism for `trfd`. X-axis represents time in window units. Each window is the average of 10^5 cycles.

curves (memory and arithmetic functional units averages) are synchronized in time and that it is possible to compare them to see how one is correlated to the other.

As it can be seen from these six figures, the utilization of the memory and functional units is certainly not uniform and follows very clear patterns that match the major loops found in the program. For example, in Fig. 4.9 it is easy to see how the program starts having a phase with almost no vector activity that lasts from window 0 to window 1300 (approx.) (this represents 1300×10^5 cycles). Then the program enters its main loop that repeats itself 5 times. This main loop can be divided in two parts: in its first quarter we have a relatively high activity in the arithmetic units (around 0.9 and 1.2 average usage) that corresponds to a relatively low activity on the bus (occupation around 0.4). The other three quarters present an 80% occupation of the bus coupled with an almost constant occupation (0.8) of the functional units. `Dyfesm` is the only program to have a very uniform shape, due to the fact that almost all its computation time is spent in a single loop. The remaining four programs behave more like `bdna` and present several phases in their execution life. Figure 4.11 shows how `arc2d` is the only program that at some points in the execution reaches a sustained occupation of its functional units close to 2. It is also one of the few programs (together with `flo52`, Fig. 4.12) that keeps the bus close to 100% occupation, although the graph shows very clear spikes where the occupation falls as low as a 40%. `Flo52` presents three clear phases with a high overall bus utilization, but with an occupation of the two arithmetic units around

0.65. The particular shape of `trfd` is due to the fact that it has two phases in the main loop in subroutine OLDA. Both phases perform a saxpy-type computation but the second one is working only on a triangular portion of the problem matrices. This triangular iteration space explains the "shrinking" phases that can be seen in Fig. 4.13.

The six graphs show that, on average, the two functional units are seldom ever used at the same time (these could also be seen if Fig. 4.3). Moreover the memory bus presents an average 20% of cycles completely idle and, many times, these free cycles correspond to phases of the program where cpu activity is highest. This implies that the available memory bandwidth should be used to either pre-advance the load of the data needed by the programs in the near future or to run another (independent) thread to try to maximize system usage.

4.6 EXECUTION COST OF VECTOR SPILL CODE

In this final section we turn now to the study of spill code. The percentage of memory traffic that is spill code (see section 3.5) is high enough to study is the effect on total execution time of the spill loads and stores. The spill load/store instructions are not part of the computation but are an overhead introduced by the compiler due to the limited size of the register file. However, spill code is a negative contribution to performance only if it actually increases the minimum number of chimes required to execute one iteration of the loop where it appears. If a loop with spill is highly compute bound, it means that there will probably be many cycles where the `LD/ST` unit is free to be used by the spill instructions. If this is the case, these spill cycles will effectively be hidden by the computation instructions. On the other hand, if a loop is memory bound, every single spill instruction will lengthen its initiation interval by VL cycles.

In this section we will study the effect of spill code on performance. There has been some previous work on the effects of spill code in vector processors. In [Lee92] it is suggested that the effect of spill code is not very bad in vector processors because spill can be hidden in those cycles where the load/store unit is not busy. Nevertheless, in [Lee92] the architecture under study was a Cray Y-MP which has the same number of functional units as our architecture but has 3 paths to memory (two load buses and one store bus) and thus has many more opportunities to hide those spill instructions. Moreover, Robbins and Robbins [RR94] shows that the occupation of these three ports is less than a 33%, indirectly confirming Lee's results

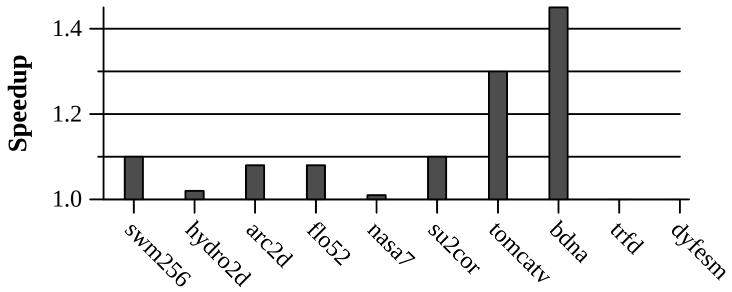

Figure 4.14 Speedup resulting from the artificial removal of all spill code.

that the spill effect is low in that configuration. We believe that in our single bus architecture the effect of spill code will be significantly higher.

To study the effects of spill on performance we have modified the simulator so that each time it finds a vector spill instruction it executes it in 0 cycles. This creates a similar effect as if we had an infinite register file, and eliminates most of the negative effect of spill code (but not all). We run the programs again with this version of the simulator and we obtain the total number of cycles needed to execute them. We can see the effect of spill code by comparing the two simulation runs.

Figure 4.14 presents the results. Each bar in the figure represents the speedup when the spill code is removed from the program as described. Program `bdna` happens to have extremely long basic blocks, coming from very long loops, and the compiler has to introduce a lot of spill code to execute them with only eight registers. We have measured the mean size of `bdna` vector basic blocks and it's close to 120 instructions (not including the scalar instructions). A similar situation appears in `tomcatv`. For the other programs, we can see a clear correlation between their percentage of spill code (see table 3.4, page 45) and the associated speedup when it is removed.

4.7 RELATED WORK

This chapter has presented several time-dependent statistics of vector programs. Most of the previous work presenting similar time-dependent micro-measurements of programs have been done on the Cray machines using its hardware performance monitor (HPM). For time-dependent data, the biggest advantage of using HPM to collect statistics is that it allows performance data to be collected while running at full speed – consequently complete applications can be measured. The disadvantage is that only those performance values implemented in HPM can be measured – for example hold issue conditions are not collected with the same precision as with our simulations. Also, only the real hardware configuration can be studied – performance differences in issue logic, functional unit latencies cannot be studied directly with HPM. Consequently, the results presented here contain much more timing-dependent data than previous work.

Vajapeyam et al. [VH92, VSH91, Vaj91] used the HPM monitor to gather statistics on hold-issue conditions and related them to the instruction types of the machine, in an attempt to understand how scalar code affected performance.

Shönauer and Häfner [SH91, SH94] present detailed measuring models of vector architectures but do not get into measurements of ILP or of hold issue conditions. Their work, however, is similar to some aspects of this chapter in that they develop analytical models that try to highlight the impact of the several latencies (memory and functional units) present in vector architectures.

Recently, Robbins and Robbins also analyzed in high detail the memory behavior of vector programs running on a Cray Y-MP [RR94]. Their work is very similar in spirit to what has been presented here. They collect time dependent data and present the evolution in time of the occupation of hardware resources. The main difference is that they put all their emphasis on the usage of the memory ports of a single Cray Y-MP. They reach conclusions very similar to ours: from a peak transfer bandwidth of 3 words per cycle (2 loads and 1 store) the machine obtains on average less than 1 word per cycle. It is remarkable the similarities of the behavior of program `trfd` running on the Y-MP (see [RR94, page 220]) and on our reference machine (see Fig. 4.13, page 74).

Most other studies have concentrated in running large benchmarks on the Y-MP and using the HPM to collect statistics of the programs with two main goals in mind: establishing a set of "average" values to be able to generate truly representative benchmarks and characterization of workload performance by application area. Examples of such types of works are Bradley et al. [BCG+91, BL81], Berry et al. [BCL91] and Dalic [Del91].

4.8 SUMMARY

In this chapter we have used the cycle level simulator of the reference architecture to analyze the behavior of the ten benchmarks.

The results show that the fraction of time executing vector computations is not as high as one would expect (around 50%). We have presented data about the fraction of utilization of the vector functional units and found that only in roughly 25% of the cycles are all vector computation units busy, while during 25% more of cycles one of the two arithmetic units is working and the last 50% of the cycles are either load/store cycles or purely scalar cycles.

The major limitation that prevents full utilization of the machine is the single bus to memory architecture, which even in the case of adding infinite number of functional units and ports, would be responsible for stalling the machine in 33.9% of all executed cycles. Related with the memory problems is the inability to chain loads and computation. This restriction is responsible for stalling the machine in a combined 25% of all executed cycles. Additional limitations found were the lack of a second functional unit able to perform multiplication and division (15.5% of executed cycles) and limited number of ports to access the vector register file (14.9%).

We have also presented the evolution in time of the instruction level parallelism found in the programs. The plots show how programs do not have a uniform pattern of instruction level parallelism and how the memory bus is, on average, idle for more than 20% of all cycles. The plots also show how typically, periods of high memory activity are coupled to periods of low cpu activity.

Finally, We also examined the impact of the vector spill code found in the programs. Since the architecture under study is single-ported, we conclude that this spill will have a clear negative effect on execution time. The negative effect of the vector spill code has been measured and found that it varies a lot but can be anywhere from a 10% up to a 45%.

5

DECOUPLED VECTOR ARCHITECTURE

Summary

In this chapter we apply the first dynamic scheduling technique, decoupling, to the baseline reference architecture. We will show the performance improvement that decoupling can achieve over an in-order vector architecture in two different ways. First, even in the absence of memory latency, the dynamic reordering of instructions improves performance. Second, when memory latency is increased, decoupling can mask most of it and make the resulting architecture almost latency-insensitive. We also present a technique that uses the input/output queues of the decoupled architectures to detect and dynamically eliminate load and store instructions, most of which are spill code.

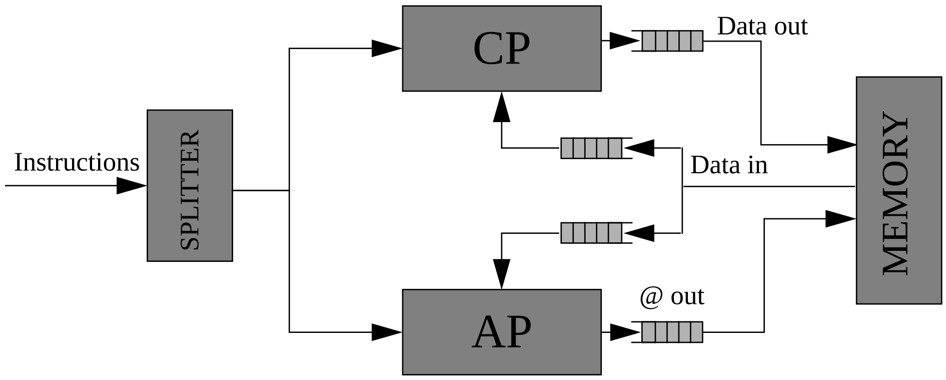

Figure 5.1 A typical Decoupled Access/Execute architecture. The Splitter divides the incoming instruction stream into memory accessing instructions, that go to the Address Processor (AP), and computation instructions that go the Computation Processor (CP). All memory accesses are performed by the AP.

5.1 INTRODUCTION

Decoupled processors attack the memory latency problem by making the observation that the execution of a program can be split into two different tasks: moving data in and out of the processor and executing all arithmetic instructions that perform the program computations [CS81, Smi82b, Smi84, SWP86]. A decoupled processor (see Fig. 5.1) typically has two independent processors, the address processor and the computation processor, that perform these two tasks asynchronously and that communicate through architectural queues [CS81, Shi82, BW83, GHL+85, SDV+87, PGH+87, Smi89]. Latency is hidden because the address processor is able to slip ahead of the computation processor and loads data that will be needed soon by the computation processor early in time [MSAD90]. This excess data produced by the address processor is stored in FIFO queues, and stays there until it is retrieved by the computation processor.

The key in the latency tolerance properties of decoupled access/execute architectures (DAE's) is that the two streams of instructions (memory and computation) do not interfere with each other. If the computation stream stalls due to a multiplier conflict, for example, this does not prevent the address processor to keep requesting data from memory. The result is that the address processor will continuously run only limited by lack of input instructions or filling of the communication queues.

Decoupling achieves a limited form of dynamic instruction scheduling [Smi89]. A sequential instruction stream is fetched and split into the two streams mentioned: memory instructions and computation instructions. Each stream proceeds out-of-order with respect to the other, but the instructions from a single stream are still executed in-order.

In the previous chapter we identified the interference between the memory and computation streams as the major source of performance degradation. In sections 4.3 and 4.4 we saw how the memory port was idle for a very large percentage of time due to two major sources of hazards: (1) an instruction waiting for data to arrive from memory blocked the dispatching of further memory instructions and (2) register file port conflicts and functional unit conflicts prevented the issue unit to advance and load data from future iterations. The two types of hazards can be summarized in one: the memory portion of a program should be executing continuously to mask latencies and ensure a continuous flow of data from main memory into the registers and functional units.

The decoupling technique gives us exactly what we need: splitting the two types of instructions we will maximize the probability that the memory pipeline is always full. The combination of vector execution and decoupling gives a very powerful latency tolerance architecture. As we will see, it is sufficient to have a small queue (buffer) holding a few vector loads to saturate the memory bus for many cycles. As long as instruction fetch and branches do not drain this load queue, the address processor will effectively use the available bandwidth while, at the same time, it will hide main memory latencies [SWP86, KHC94]. In contrast, scalar decoupled architectures can not tolerate latencies as large as decoupled vector architectures and might need the help of data caches [KHC94].

This chapter will show that using decoupling techniques in a vector processor, the performance of vector programs can be greatly improved [EV96]. We will show how, even for an ideal memory system with zero latency, decoupling provides a significant advantage over standard mode of operation. We will also present data showing that for more realistic latencies, decoupled vector architectures perform substantially better than non-decoupled vector architectures. Although we only look at the single processor case, the decoupling technique would also be very effective in vector multiprocessors to help reducing the negative effect of conflicts in the interconnection network and in the memory modules.

5.2 THE DECOUPLED VECTOR ARCHITECTURE (DVA)

The decoupled vector architecture (DVA) we propose uses a fetch processor (*FP*) to split the incoming, non-decoupled, instruction stream into three different decoupled streams (see Fig. 5.2). Each of these three streams goes to a different processor: the address processor (*AP*), that performs all memory accesses on behalf of the other two processors, the scalar processor (*SP*), that performs all scalar computations and the vector processor (*VP*), that performs all vector computations. The three processors communicate through a set of *implementational* queues and proceed independently. This set of queues is akin to the implementational queues that can be found in the floating point part of the R8000 microprocessor [Hsu94]. The main difference of this decoupled architecture with previous scalar decoupled architectures such as the ZS-1 [SDV+87], the MAP-200 [CS81], PIPE [GHL+85] or FOM [BW83], is that it has *two* computational processors instead of just one. These two computation processors, the *SP* and the *VP*, have been split due to the very different nature of the operands on which they work (scalars and vectors, respectively).

The fetch processor fetches instructions from a sequential, non-decoupled instruction stream and translates them into a decoupled version. The translation is such that each processor can proceed independently and, yet, synchronizes through the communication queues when needed. For example, when a memory instruction that loads register v5 is fetched by the *FP*, it is translated into two pseudo-instructions: a load instruction, which is sent to the *AP*, that will load data into the vector load data queue (VLDQ, queue no. 1 in Fig. 5.2), and a `qmov` instruction, sent to the *VP*, that dictates a move operation between the VLDQ and the final destination register v5. It is important to note that the `qmov`'s generated by the FP are not "instructions" in the real sense, i.e., they do not belong to the programmer visible instruction set. These `qmov` opcodes are hidden inside the implementation.

Note that the total hardware added to the original reference architecture shown in Fig. 2.1, page 26, consists only of the communication queues and a private decode unit for each one of the three processors. The resources inside each processor are the same in the decoupled vector architecture and in the reference architecture. It is worth noting, though, that while most queues added are scalar queues and, therefore, require a small amount of extra area, the VLDQ and VSDQ hold full vector registers (queues 2 and 3 in Fig. 5.2). That is, each slot in these queues is equivalent to a normal vector register of 128 elements, thus requiring 1Kb of storage space. One of the key points in this architecture will be to achieve good performance with relatively few slots in these two queues.

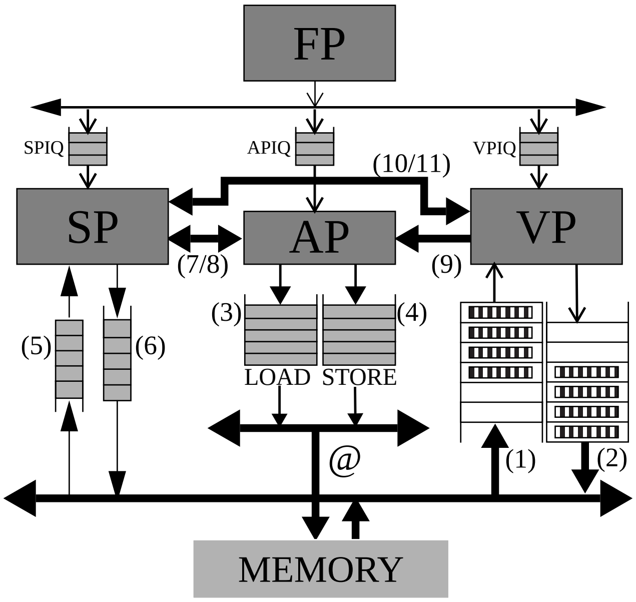

Figure 5.2 The decoupled vector architecture. Queue names: (1) vector load data queue –VLDQ, (2) vector store data queue –VSDQ, (3) address load queue –ALQ, (4) address store queue –ASQ, (5) scalar load data queue –SLDQ, (6) scalar store data queue –SSDQ, (7/8) Scalar processor to Address processor Control Queues –SACQ and ASCQ, (9) Vector processor to Address processor Control Queue –VACQ, (10/11) Scalar processor to Vector processor Control Queues –SVCQ and VSCQ.

The address processor performs all memory accesses, both scalar and vector, as well as all address computations. Scalar memory accesses go first through a scalar cache that holds only scalar data. Vector accesses do not go through the cache and access main memory directly. There is only one pipelined port to access memory that has to be shared by all memory accesses. The address processor inserts load instructions into the Address Load Queue (ALQ) and store instructions into the Address Store Queue (ASQ). Stores stay in the queue until their associated data shows up either in the output queue of the *VP* (the vector store data queue –VSDQ), or in the output queue of the *SP* (the scalar store data queue –SSDQ). When either a load or a store becomes ready, i.e, it has no dependencies and its associated data, if necessary, is present, it is sent over the address bus as soon as it becomes available. In the case of having both a load and a store ready, the *AP* always gives priority to loads.

To preserve the sequential semantics of a program, the address processor needs to ensure a safe ordering between the memory instructions held in the ALQ and ASQ. All memory accesses are processed in two steps: first, their associated "memory region" is computed. Second, this region is used to disambiguate the memory instruction against all previous memory instructions still being held in the address queues of the *AP*. Using this disambiguation information, a dependency scoreboard is maintained. This scoreboard ensures that (1) all loads are executed in-order, (2) all stores are executed in-order and (3) loads can execute before older stores if their associated memory regions do not overlap at all. When dependences are found, the scoreboard guarantees that loads and stores will be performed in the original program order so that correctness is guaranteed.

A "memory region" is defined by a 6-tuple: $\langle @_1, @_2, \texttt{vl}, \texttt{vs}, \texttt{sz}, \texttt{v} \rangle$. Virtual addresses $@_1$ and $@_2$ are the start and end addresses, resp., of a consecutive region of bytes in memory, and vl, vs, and sz are the vector length, vector stride and access granularity needed by vector memory operations. The end address, $@_2$, is computed as $@_1 + (vl - 1) * vs + sz$. For scalar memory accesses, vl is set to 1 and vs to 0. For the special case of gathers and scatters, which can not be properly characterized by a memory region, $@_1$ is set to 0 and $@_2$ is set to $2^{32} - 1$, so that the scoreboard will find a dependence between a gather/scatter and all previous and future memory instructions.

The vector processor performs all vector computations. The main difference between the *VP* and the reference architecture is that the *VP* has two functional units dedicated to move data in and out of the processor. These two units, the QMOV units, are able to move data from the VLDQ data queue (filled by the *AP*) into the vector registers and move data from the vector registers into the VSDQ (which will be drained by the *AP* sending its contents to memory). We have included *two* QMOV units instead of one because otherwise the *VP* would be paying a high overhead in some very common sequences of code, when compared to the reference architecture.

The set of control queues connecting the three processors, queues 7–11 in Fig. 5.2, are needed for those instructions that have mixed operands. The most common case is in vector instructions, that can have a scalar register as a source operand (i.e., `mul v0,s3 -> v5`). Thus a FIFO queue connecting the *SP* and the *VP* is necessary (the SVCQ, queue number 10 in Fig. 5.2) to communicate register values between the scalar and the vector processor. As it happens for memory operations, the splitter is in charge of generating a `qmov` pseudo-instruction for the *SP* that will put the required register into the SVCQ and a modified vector instruction that will read the value from the SVCQ at the right time. The C3400 has two vector reduction instructions that are executed in the *VP* but that produce a single result that must be stored in the S register file. This forces the inclusion of a queue connecting the *VP* to the *SP*, the vector processor to scalar processor control queue (VSCQ, queue number 11 in Fig. 5.2). There are two control queues connecting the *AP* and the *SP* for very similar reasons. The C3400 has both S-type instructions that accept an A-register as an operand, and A-type instructions that accept an S-register as an operand. Thus there is data movement in both directions between the two processors. Finally, for gathers and scatters, the *VP* needs to communicate a vector filled with memory addresses to the *AP*. This is the motivation for the VACQ queue shown in Fig. 5.2 (queue number 9).

5.3 PERFORMANCE OF THE DVA

Our goal is to show the performance advantages of the DVA over the reference machine in two different aspects: first, even with an ideal memory system without latency, the limited dynamic scheduling performed by the DVA improves upon the sequential in-order execution model of the reference machine. Second, when memory latency is increased, the decoupled vector architecture suffers a minimal performance impact, while in the reference machine, as we have seen in section 4.2, performance significantly degrades.

We will start by looking at the maximum performance advantages that a decoupled vector architecture can have over a traditional in-order machine. To do so, we will start with a very idealized version of the DVA in which all queues have a very large capacity and where the fetch processor has a perfect branch prediction mechanism. Although such a machine is by no means realizable, it will give us an upper bound of the expected improvements that decoupling can yield. This machine will be referred to as the unbounded decoupled vector architecture (UDVA).

The next step will be to define a realistic version of this idealized machine. The problem is that there are many parameters that have to be fixed to reach a reasonable implementation. In particular, one must decide all queue lengths, all instruction buffer lengths and the complexity of a real branch predictor. Instead of choosing some ad-hoc values for all these parameters and comparing the resulting machine against the reference architecture, we will try to fix each parameter in a “cascaded” way. We will proceed step by step and fix only one parameter at each step. This will help pointing out the relative impact of each choice in the final performance of our realistic machine. Admittedly, the order in which we study and fix the different parameters does affect the relative contribution of each parameter to overall performance. Nonetheless, the order chosen tries to reflect a reasonable decision-making process found in all machine designs: first look at those parameters that will constrain all other decisions. For example, the size of the instruction queues of each of the three processors (*VP*, *SP* and *AP*) determines some practical bounds on the maximum occupation of all other queues in the system. Thus, this will be the first parameter under examination. Once all parameters are fixed we will have a realistic version of the unbounded machine, which we will refer to as the RDVA.

5.3.1 UDVA performance

The Unbounded DVA architecture (UDVA) is a version of the decoupled architecture that has all of its queues set to a very large value (128 slots) and no latency delays (neither memory latency nor functional unit latencies). Moreover, a perfect branch prediction model is assumed. The I-cache is not modeled in any of the following experiments, since our previous data indicates a very low pressure on the I-cache [EV95]. All arithmetic functional units, both scalar an vector, have a 1 cycle latency. The vector register file read and write crossbars have no latency and there is no startup penalty for vector instructions.

The benefits of decoupling can be seen in Fig. 5.3. For each program we plot the total execution time of the UDVA and the REF architectures when memory latency is varied between 1 and 100 cycles. In each graph we also show the minimum absolute execution time that can theoretically be achieved (curve "IDEAL", along the bottom of each graph), as already discussed in section 4.2, page 58 and in section 3.6.

The overall results suggest two important points. First, the UDVA architecture shows a clear speedup over the REF architecture even when memory latency is just 1 cycle. Even if there is no latency in the memory system, the decoupling produces a similar effect to a prefetching technique, with the advantage that the *AP* knows which data has to be loaded (no incorrect prefetches). The second important point is that the slopes of the execution time curves for the reference and the decoupled architectures are substantially different. This implies that decoupling tolerates long memory delay much better than current vector architectures.

Overall, decoupling helps to minimizing the number of cycles where the machine is halted waiting for memory. Recall from section 4.3 that the execution time of the program could be

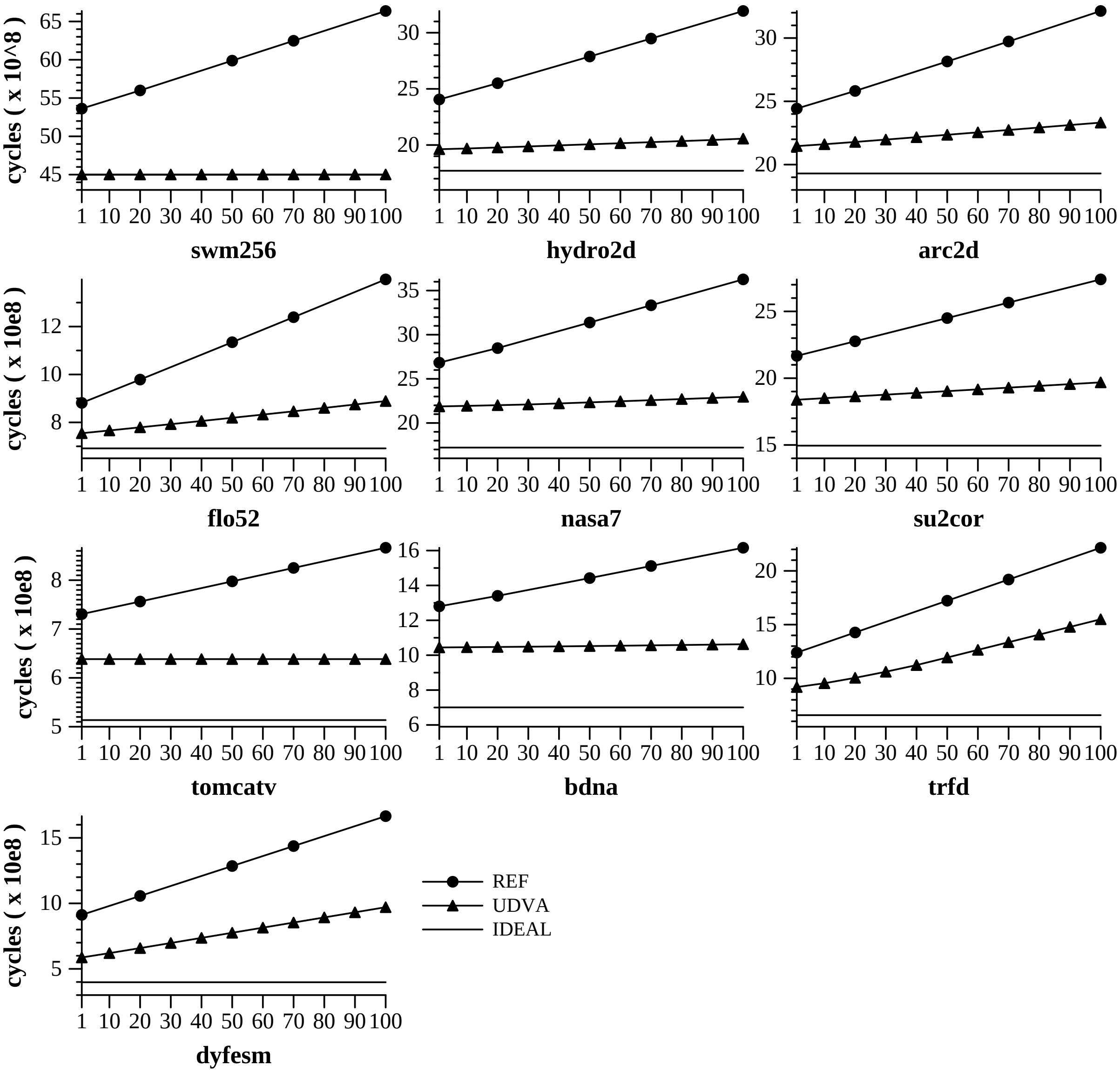

Figure 5.3 UDVA versus Reference architecture for the benchmark programs.

partitioned into eight different states. Decoupling greatly reduces the cycles spent in state $\langle \; , \; , \; \rangle$.

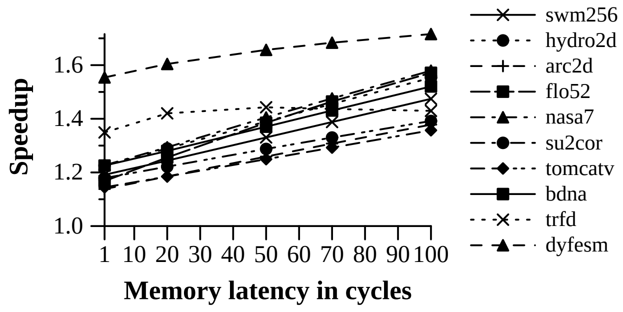

Figure 5.4 Speedup of the DVA over the Reference architecture for the benchmark programs.

The two programs that have less decoupling are `trfd` and `dyfesm`. As it can be seen, the slope of the UDVA curves for them is not as flat as it is for the other programs.

To summarize the speedups obtained, Fig. 5.4 presents the speedup of the UDVA over the REF architecture for each particular value of memory latency. Speedups (at latency 100) range from a 1.32 for `tomcatv` to a 1.70 for `dyfesm`.

5.3.2 Reducing Instruction Queue Length

The first limitation we introduce into the UDVA is the reduction of the instruction queues (IQ) that feed the three computational processors (*AP*, *SP*, *VP*). In this section we look at the slowdown experienced by the UDVA when the size of the APIQ, SPIQ and VPIQ is reduced from 128 instructions to 32, 16, 8 and 4 instructions only. In order to reduce the amount of simulation required, we have chosen to fix the value of the memory latency parameter at 50 cycles. As we have seen in the previous section, the UDVA tolerates very well a wide range of memory latencies. Thus we expect this value of 50 cycles to be quite representative of the full 1–100 latency range.

The size of the instruction queues is very important since it gives an upper bound on the occupation of all the queues in the system. For example, it determines the maximum number of entries that we can have waiting in the load address queues.

Figure 5.5 presents the slowdown with respect to the UDVA for our ten benchmarks when the three instruction queues are reduced to 32, 16, 8 and 4 slots. From Fig. 5.5 we can see

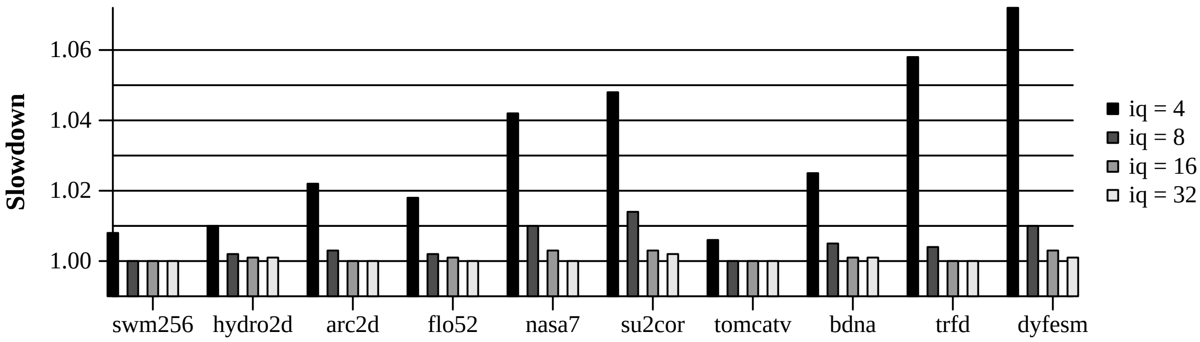

Figure 5.5 Slowdown experienced by UDVA when reducing the IQ size.

that the performance for 128-, 32- and 16-entries instruction queues is virtually the same for all benchmarks. From these numbers, we decided to set the IQ length to 16 entries for the rest of experiments presented in this chapter. This size is in line with the typical instruction queues found in current microprocessors [Yag96].

5.3.3 Effects of branch prediction

In this section we look at the negative effects introduced by branch mispredictions. The branch prediction mechanism evaluated is a direct-mapped BTB holding for each entry the branch target address and a 2-bit predictor (the predictor found in [LS84]). We augmented the basic BTB mechanism with an 8-deep return stack (similar to the one found in [BBB+95]).

We evaluated the accuracy of the branch predictor for BTB's having 16, 32 and 64 entries. The accuracy varies a lot across the set of benchmarks, as is shown in Fig. 5.6. Programs `flo52` and `nasa7` came out with the worst misprediction rate (around 30%) while `tomcatv` has less than 0.4% of mispredicted branches. Nonetheless, the misprediction rate is rather high for a set of programs that are considered to have an "easy" jumping pattern (numerical codes tend to be dominated by DO-loops). This is due to the combination of two factors: first, vectorization has reduced the absolute total number of branches performed by the programs in an unbalanced way. The number of easily-predictable loop branches has been diminished by a factor that is proportional to the vector length (could be as high as 128) while the difficult

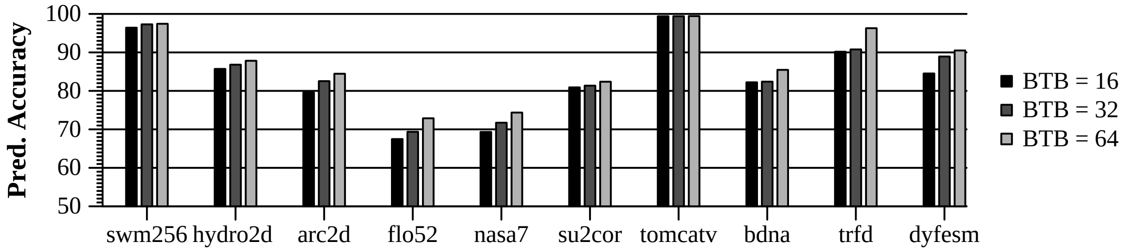

Figure 5.6 Branch prediction accuracy for three different BTB sizes.

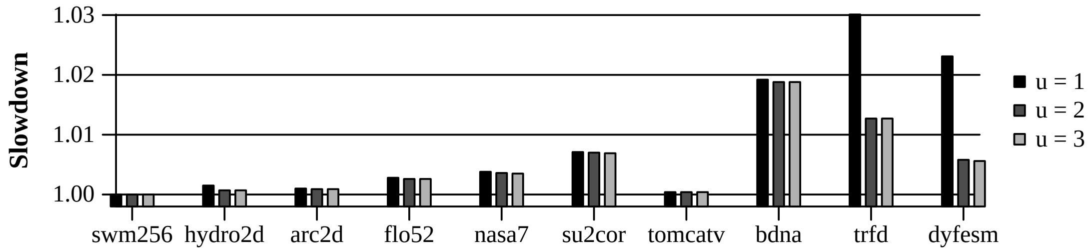

Figure 5.7 Slowdown due to branch mispredictions for three models of speculation.

branches found in the remaining scalar portion of the code are essentially the same. The second factor is that we are using a very small BTB compared to what can be found in current superscalar microprocessors, where a typical BTB could have up to 4096 entries [TGN95].

Although the prediction accuracy is not very good, the impact of mispredicted branches on total execution time is very small. Figure 5.7 presents the slowdown due to mispredicted branches relative to the performance of the architecture from section 5.3.2. Since the prediction accuracy was not very high, we tested the benefit that could be obtained by being able to speculate across several branches. In Fig. 5.7 the bars labeled "u=1" correspond to an architecture that only allows one unresolved branch. Bars labeled "u=2" and "u=3" correspond to being able to speculate across 2 and 3 branches respectively.

A first observation is that the vector execution model enhanced with decoupling tolerates very well mispredicted branches. For example, while `flo52` has a 30% misprediction rate, the total impact of those mispredicted branches is below 0.5%. Also, there is not a strong correlation between misprediction rate and execution time impact. Program `bdna` is the one suffering most from branches (almost a 2% impact) and yet has a far better prediction accuracy than either `flo52` or `nasa7`. This behavior is due to a highly scalar portion of the execution of `bdna` (at the beginning) that concentrates most of the mispredictions (in the vector portion the prediction accuracy is around 90%) and that also concentrates most of the impact of those mispredictions.

A second observation is that while speculating across multiple branches provides some benefits, specially for `dyfesm`, its cost is certainly not justified. The simplicity of only having one outstanding branch to be resolved is a plus for vector architectures.

All the simulations in the following sections have been performed using a 64-entry BTB and allowing only 1 unresolved branch.

5.3.4 Reducing the vector queues length

Vector Load Data Queue

This section will look at the usage of the vector load data queue. The goal is to determine a queue size that achieves almost the same performance as the 128-slot queue used in the previous sections and yet minimizes as much as possible hardware costs. Recall that each slot in one of the vector queues consumes 1Kb of space and, thus, an 8 element queue would take more or less the same space as a small cache.

Figure 5.8 presents the distribution of busy slots in the VLDQ for the benchmark programs. For each program we plot three distributions corresponding to three different memory latency values. Each bar in the graphs represents the total number of cycles that the VLDQ had a certain number of busy slots For example, for `trfd` at latency 1, the VLDQ was completely empty (zero busy slots) for around 500 million cycles.

From Fig. 5.8 we can see that it is not very common to use more than 6 slots. Except for `swm256` and `tomcatv`, 6 slots are enough to cover around 85–90% of all cycles. When latency is increased from 1 cycle to 50 and 100 cycles, the graphs show a shift of the occupation towards higher number of slots. As an example, consider programs `arc2d`, `nasa7` and `su2cor`. For 1 cycle memory latency, these programs have typically 2–3 busy slots. When latency increases, these three programs show an increase in the total usage of the VLDQ, and they typically

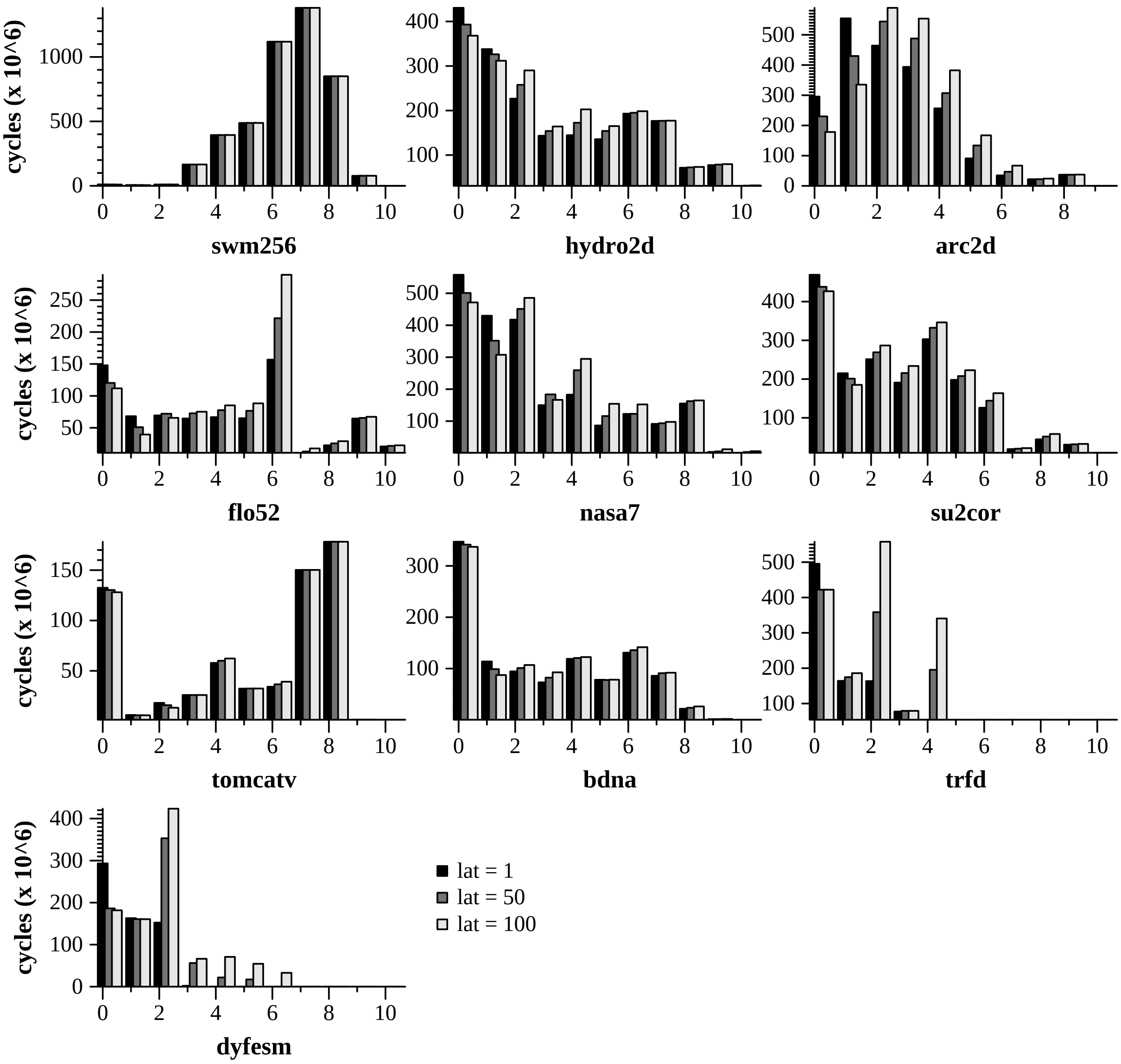

Figure 5.8 Busy slots in the VLDQ for the benchmark programs for three different memory latency values.

use around 4–5 slots. As expected, the longer the memory latency, the higher the number of busy slots, since the memory system has more outstanding requests and, therefore, needs more slots in the queue.

For two programs (`trfd` and `dyfesm`) there is a sharp peak at 2 slots when memory latency is set to 100 cycles. Analysis of the main loops of these two programs reveals that they are highly memory bound. When the DVA is executing a memory bound loop, what happens is that the *VP* executes faster than the *AP*. In the steady state, the *VP* will most of the time be waiting for data to arrive to its input queue (the VLDQ). As soon as one vector is ready in the VLDQ, the *VP* will execute a `qmov` instruction and start moving it to a vector register. At the same time the *AP* will most probably start another vector load. Thus, in this steady state, it is typical to have two busy slots in the VLDQ.

Program `flo52` experiences a similar situation, having a sharp peak for 6 slots. This program has an average vector length around 50 elements. This means that, in our non-memory-chaining environment with a latency of 100 cycles, the *AP* will be able to request three vectors roughly before the last element of the first vector requested arrives. Moreover, a close study of the basic blocks that take most of the time show a couple of basic blocks that have an average vector length of 17 elements. Now, in a 100 cycles frame, the *AP* can request 6 17-elements-long vector before the last element of the first vector requested returns to the processor. The two situations mentioned, combined with the fact that, in general, decoupling manages to bring data ahead of time, mostly explain the sharp increase at the 6 slots mark.

The queue length seems to be bounded by 10 slots, with none of the programs having at any point in time more than that amount of full slots. Since both the address queues and the vector data queues we are using at this stage are 128 elements long, this a counterintuitive result. One would expect that, for compute bound loops, the VLDQ would be completely filled, since the *AP* always runs ahead of the *VP*. It turns out that when a loop is compute bound, the resource that becomes the bottleneck is the VPIQ. The VPIQ is limited to 16 slots, and for each memory operation the *FP* will insert a QMOV pseudo-instruction into the VPIQ. Thus, a compute bound loop will be able to hold a maximum of 9 computation instructions and 7 `qmov`'s in the VPIQ. Once the VPIQ is full, the *FP* will block and the *AP* will not be able to get more vector loads. As a result, the *AP* will not be able to insert more than eight elements in the VLDQ. Since there are two QMOV units in the vector processor the total maximum full slots is 10: the eight already mentioned plus two slots in flight from the queue into the vector registers. It is worth pointing out that this behavior will not affect final performance since, if the loop is compute bound, the maximum speed is determined by the *VP* and not by the *AP*.

The execution impact of reducing the VLDQ size can be seen in Fig. 5.9. The slowdown presented is with respect to the UDVA of the previous section. As expected from the data

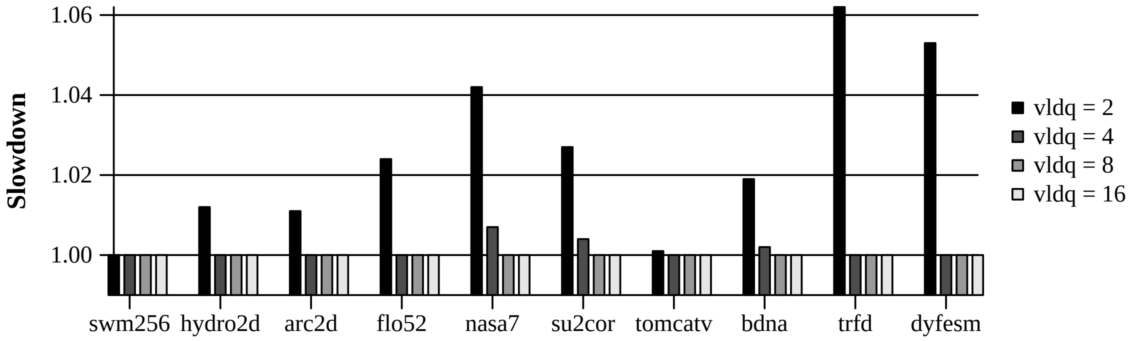

Figure 5.9 Slowdown due to reducing the VLDQ size (relative to section 5.3.3).

seen in Fig. 5.8, reducing the queue size to 16 or 8 slots is not noticeable for most programs. Going down to 4 slots affects mostly `nasa7` and `bdna` but the impact is less than a 1%. Further reducing the VLDQ to 2 slots would start to hurt performance although not very much. The worst case would be for `trfd` and `dyfesm` with a 5% and 6% impact, mostly due to the peculiar distribution of their VLDQ occupation already mentioned. As we have already discussed, 2 slots is clearly a lower bound on the size of the VLDQ to accommodate most memory bound loops. Reducing that queue to 1 slot would stop most of the decoupling effect present in the architecture.

Looking at all the data presented in this section we decided to pick a 4 slot VLDQ. All following sections use this size for the VLDQ.

Vector Store Data Queue

The usage of the vector store data queue (VSDQ) presents a very different pattern from the VLDQ. Recall that the *AP* always tries to give priority to load operations in front of stores. This has the effect of putting much more pressure on the VSDQ, which can, at some points, become full (even with 128 slots!). This situation is not as unusual as it may seem. As long as the *AP* encounters no dependencies between a load and a store and as long as there are loads to dispatch, no stores will be retrieved from the VSDQ and sent to memory. Thus the occupation of the VSDQ is much higher than that of the VLDQ.

Figure 5.10 presents the distribution of busy slots in the VSDQ for the benchmark programs. As we did for the load queue, we plot three distributions corresponding to three different

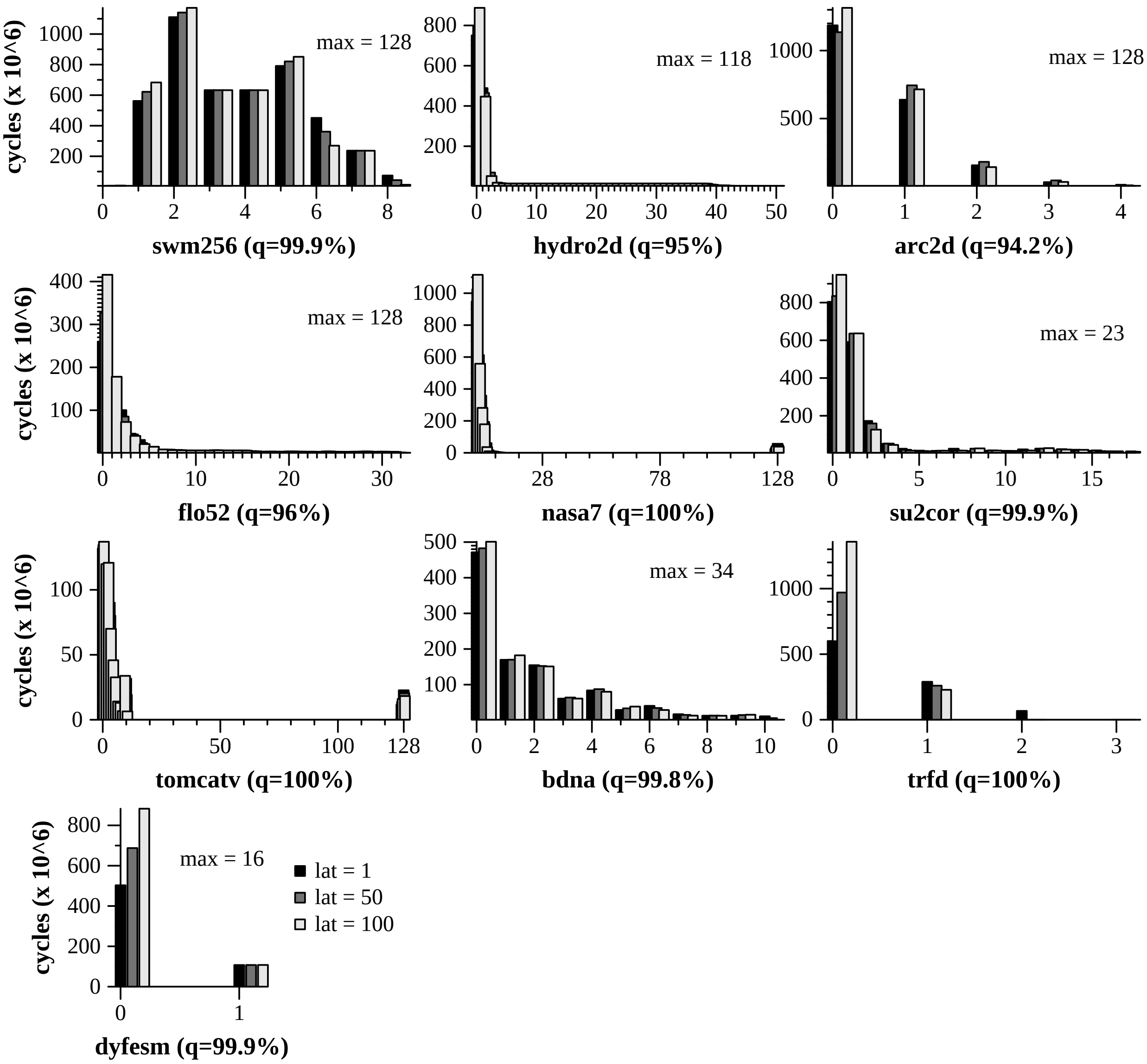

Figure 5.10 Busy slots in the VSDQ for the benchmarks for three different memory latency values.

memory latency values. Each bar in the graphs represents the total number of cycles that the VSDQ had a certain number of busy slots. To make the plots more clear, we have pruned some of the graphs. Next to the name of each program we indicate the quartile amount being shown. For example, the full set of data is shown for `nasa7` (q=100%) while the bars in the `hydro2d` graph only present 95% of all available data (the rest of the data set was too small to be seen on the plot). To compensate for this loss of information, each graph also includes the maximum value that the X axis took for that particular program. Again, for `hydro2d`, the graph shows that the maximum occupation of the VSDQ reached 118 slots, although the X-axis of the plot only goes up to 50.

Note that for 6 of the programs, at some point the queue was completely filled (128 full slots), although the most common occupation ranges between 0 and 10 slots. What are the situations that lead to a complete occupation ? Mainly two cases. First, loops with no memory dependencies between loads and stores. For those loops, yet to be stored data is held in the VSDQ until the queue completely fills up. Then, for each new store wishing to enter the VSDQ, the *AP* is forced to send the oldest store in the queue to memory. Thus, a steady state is reached. This situation is not that uncommon, as the `nasa7` and `tomcatv` programs show. For program `nasa7`, the bars at the 127 and 128 mark add up to 3.7% of all executed cycles, and for `tomcatv` they represent 5.4%. The second case is actually an extension of the first. When the program exits a loop nest and proceeds to the following one, it is common for the two loops to access disjoint data structures. This implies that there will be little or no conflicts between the two loop nests store instructions and, thus, the store data belonging to the first will remain for a long period in the queue while the second loop is executing.

For the other 4 programs, the occupation of the queue is bounded. For `bdna`, with a maximum occupation of 34 slots in the queue and `su2cor` (maximum 23), the bounding is mostly due to their high percentage of spill code. Each time a vector load tries to recover a vector from the stack previously spilled by a store, the *AP* detects a dependency and needs to update the contents of memory draining the queue. This heavily limits the amount of old stores that are kept in the VSDQ.

Programs `trfd` and `dyfesm` are qualitatively different. These two program happen to have recurrences in their main loops that limit the amount of slip-ahead of the *AP* with respect to the *VP*. Program `dyfesm`, for example, has a recurrence that forces the three main processors, the *AP*, the *SP* and the *VP* to work in lock step, thus typically allowing only a maximum of 1 full slot. Program `trfd` has at its core a triangular matrix decomposition. The order in which the matrix is accessed makes each iteration of the main loop dependent on some of the previous iterations, which causes a lot of load-store dependencies in the queues. These dependencies are resolved, as in the case for spill code, by draining the queue and updating memory. Thus, the queue never reaches a large occupancy. We note that these two programs

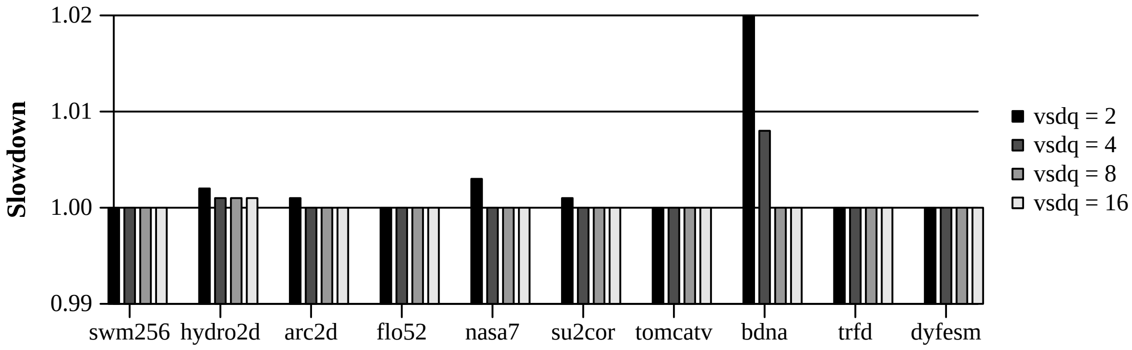

Figure 5.11 Slowdown due to reducing the VSDQ queues size (relative to section 5.3.4).

achieve good speedups (see Fig. 5.4) although they actually see their performance degraded when memory latency is increased, due to the recurrences just mentioned.

The execution impact of reducing the VSDQ size can be seen in Fig. 5.11. The bars show that the amount of storage in the VSDQ is not very important to performance. This is mostly due to the fact that we are in a single-memory port environment. No matter how we reorder loads and stores between themselves, every single store will have to be performed anyway. Thus, sending a store to memory at the point where its data is ready or later does not change much the overall computation rate.

From all the data presented in this section we selected the 4-slot VSDQ for all following experiments.

5.3.5 Reducing the scalar queues

In this section we will look at the impact of reducing the size of the various scalar queues in the system. Looking back to Fig. 5.2 we will be reducing queues numbered 3–8 and 10–11 from 128 slots down to 16, 8, 4, and 2 slots. Queue number 9, the VACQ, *VP*-to-*AP* Control queue, will be reduced from 128 slots to just 1 slot. Note that this queue holds one full vector register used in gather/scatter operations.

The impact of those reductions can be seen in Fig. 5.12. Overall, using an 8 entry queue for all the scalar queues is enough to sustain the same performance as the 128 entry queues.

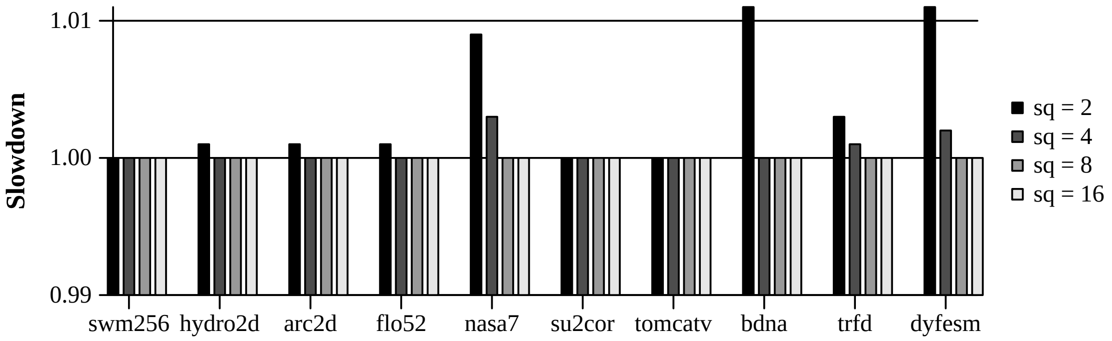

Figure 5.12 Slowdown due to reducing the Scalar queues size (relative to section 5.9).

Even for small 2 entry queues, the slowdown is around 1.01 for only 3 programs, `dyfesm`, `bdna` and `nasa7`. Nonetheless, it is worth noting that our programs are heavily vectorized. A small degradation in performance on the scalar side is tempered by the small percentage of scalar code present in our benchmarks. In order to make a safe decision we took a 16 entries queue for all the scalar queues present in the architecture. The 16 slot size is chosen because it is reasonably close to what modern out-of-order superscalar processors have in their queues [Yag96].

5.3.6 Effects of functional unit latencies

In this section we will look at the effects of latencies inside the computational processors of our architecture. So far, all models simulated had all of their functional units using a 1 cycle latency and the vector registers read/write crossbars were modeled as if it was free to go through them. This section will proceed in three steps.

First we will add to our architecture the latency of the vector functional units. Table 2.2, page 28, shows the values chosen. In a second step, we will add a penalty of 1 cycle of vector startup for each vector operation. In a third step we will add 2 cycles of vector read crossbar latency and then we will add 2 cycles of vector write crossbar latencies. In the last step, we will set the latencies of the scalar units to those shown in table 2.2.

Figure 5.13 shows as a set of stacked bars, the degradation in performance as each of the aforementioned effects is added. The bar at the bottom, labeled “vect. lat” represents the

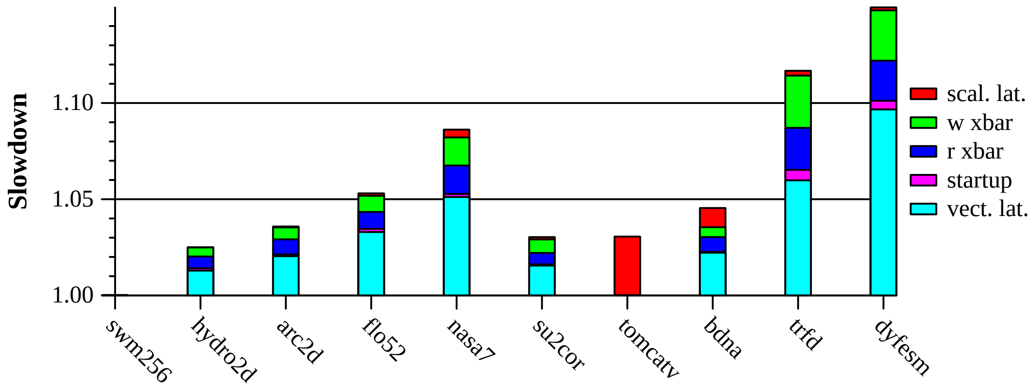

Figure 5.13 Slowdown due to modeling arithmetic unit latencies and vector pipeline crossbars. (relative to section 5.3.5).

slowdown relative to section 5.3.5. The following bar, labeled "startup", is the slowdown with respect to the performance of the "vect. lat" bar. Similarly for each of the following bars. Thus, the total height of each bar is the combined slowdown of all these effects.

Figure 5.13 shows two different behaviors. For seven out of the ten programs, latencies have a very small impact (below 5%). This is due to the fact that decoupling is not only good for tolerating memory latencies but, in general, they help in covering up the latencies inside the processor. On the other hand, two programs, `trfd` and `dyfesm` show slowdowns as bad as 1.11 and 1.15, respectively.

The behavior of these two programs is not surprising given what we already saw in section 5.3.4. Both `trfd` and `dyfesm` have difficulties in decoupling because of the inter-iteration dependences of `trfd` and the recurrences in `dyfesm`. As we saw, both of them only achieve a very small occupation of the vector load data queue (VLDQ) which indicates a bad degree of decoupling. If we couple this fact with the relatively low vector lengths of `trfd` and `dyfesm`, we see that any cycle added to the vector dependency graph is typically enlarging the critical path of the program.

Going into the detailed breakdown, Fig. 5.13 shows that the vector functional unit latencies have the highest impact of all latencies added in this section. It is worth noting, though, that the order in which the latencies are added might have some impact on the relative importance of each individual category. Nonetheless, since the vector latencies units are the largest of the latencies added and since the programs are highly vectorized they are the group that

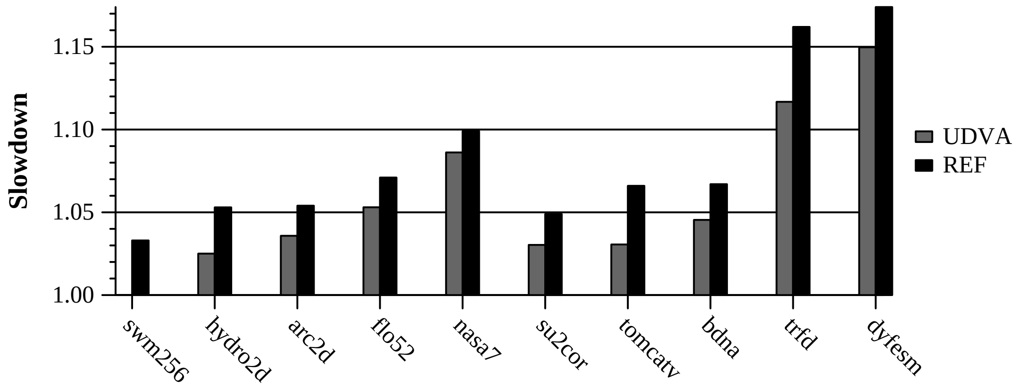

Figure 5.14 Comparison of functional unit latency impact between the UDVA and REF machines.

most likely will impact performance, as Fig. 5.13 confirms. The startup penalty is only seen in programs `trfd` and `dyfesm`, where its impact is less than a 1%. The vector register file read/write crossbar latencies have an impact on all programs, except, `swm256` and `tomcatv`. Typically both latencies have the same amount of impact, being between 0.5 and 1 percentage points for the most vectorized codes and around 2–3 percentage points in the less vectorized `trfd` and `dyfesm`. The scalar latencies have a low impact on all programs, partly due to them being shorter than the vector ones and partly due to the small fraction of scalar code.

We decided to compare the impact of functional unit latencies in the reference machine and in the DVA machine. To do so, we simulate a reference machine with no latencies at all and a reference machine with the standard latencies and compute the resulting slowdowns. Then we compare these slowdowns to the slowdowns of Fig. 5.13, which we just presented. The result of the comparison can be seen in Fig. 5.14. The results show that in all cases the effect of functional unit latencies is much worse in the in-order reference machine that in the decoupled machine. Since decoupling introduces some form of dynamic scheduling, it can hide latencies that were previously in the critical path by performing memory loads in advance.

5.3.7 RDVA versus REF

With the data presented in the last section, we have reached a realistic implementation of the originally proposed UDVA. This realistic version will be referred to as RDVA and its main parameters are as follows: all instruction queues and scalar queues are 16 entries long. The address queues in the *AP* are also 16 entries long. The latencies used in the functional units and in the read/write crossbars of the register file are those shown in table 2.2. The VLDQ and VSDQ each have 4 slots, and the control queue connecting the *VP* and the *AP* has a single slot. The branch prediction mechanism is a 64 entry BTB with at most 1 unresolved branch being supported.

In this section we will we re-plot a full-scale comparison of UDVA, RDVA and REF for several latencies. Figure 5.15 presents the data for the three architectures when memory latency is varied from 1 cycle to 100 cycles. For almost all programs, the difference between the UDVA and RDVA is rather small, and their slopes are relatively parallel. For `swm256`, the difference is almost 0. For programs `hydro2d`, `tomcatv`, `arc2d` and `su2cor`, the slowdown between RDVA and UDVA is less than 1.05 (respectively, it is 1.029, 1.031, 1.037 and 1.044). Programs `flo52`, `bdna` and `nasa7` have a higher slowdown and, moreover, the slope of the curve for the RDVA performance starts diverging from the UDVA at high values of latency. Finally, `dyfesm` and `trfd`, as seen in previous sections, take a significant performance hit when going from the UDVA to the RDVA.

5.4 DYNAMIC LOAD/STORE ELIMINATION

While decoupling has reduced the memory latency problem, it has not addressed the problem of the extra spill code present in our applications. The decoupling technique has added more "hidden" vector registers organized as a set of two queues, the VLDQ and the VSDQ. For example, the final RDVA machine has a total vector storage capacity equivalent to 16 vector registers: 8 true vector registers and 4 slots in each queue. Nonetheless, these extra registers do not show in the logical register file space and, therefore, there is no obvious way to take advantage of them to reduce the amount of spill code that the machine has to execute.

This section will present a dynamic technique to take advantage of the extra slots in the queue to capture some of the spill code and avoid having to access main memory for these particular memory operations. We will exploit our knowledge of the typical structure of spill code (see section 3.5.1, page 46) to detect at run-time memory operations that are spill code and eliminate them.

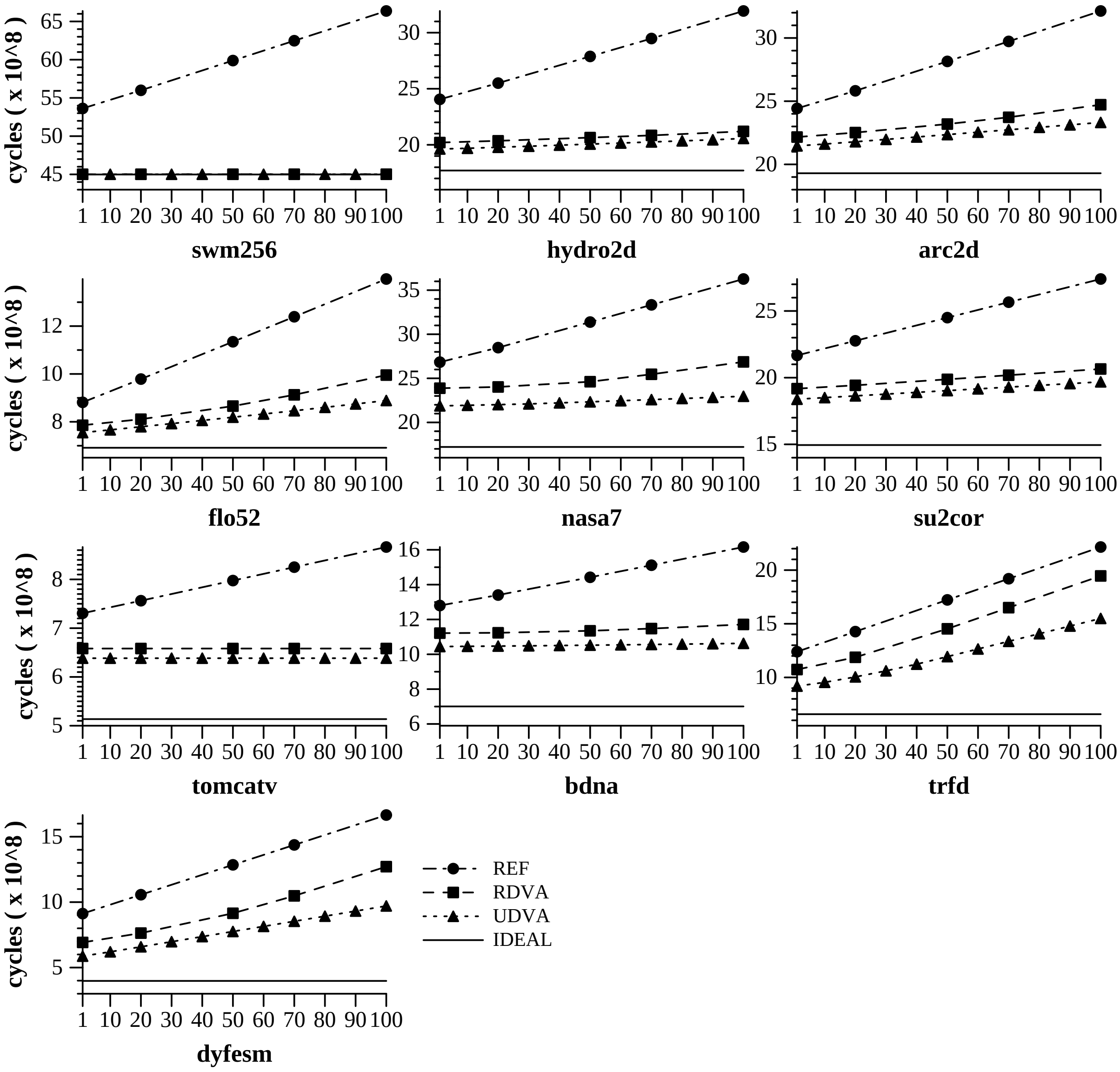

Figure 5.15 Comparison of REF, UDVA and RDVA execution times for several latencies.

5.4.1 Detection of redundant memory operations

When a vector load instruction is inserted in the address load queue (ALQ), it is disambiguated against all stores present in the address store Queue (ASQ). If a dependency is found with a certain store, the store queue contents must be sent to main memory before proceeding with the load. An interesting possibility is that the load might be *identical* to some store in the store queue. In such a case, when the data for the store becomes available, the data can be bypassed from the VSDQ back into the VLDQ and the load eliminated.

This bypassing is a limited form of data caching and has several advantages. First, the data being bypassed does not suffer any memory latency penalties. Second, during the bypass operation the memory port is idle and can be used by subsequent independent vector memory operations. In this second case, bypassing gives the illusion of having two memory ports, since two different vectors are being moved into the VLDQ simultaneously.

In a similar way, when a store instruction is inserted into the address store queue, it can be disambiguated against all previous stores. If an *exact* dependency is found between two stores without any other in-between store being affected and without any in-between load depending on the fist store being present, the first (oldest) store can be eliminated without going to main memory. This elimination also results in a net reduction of total traffic.

When will all these types of load/store matching and elimination occur ? Generally speaking, we have several different cases:

- Matchings between memory operations belonging to the same iteration of a loop. This is the classical example of spill code, where a vector register is saved to the stack and later reloaded typically within the same basic block.
- Matchings between memory operations belonging to different iterations of a loop. This types of matchings are the result of short inter-iteration dependences or the result of caching effects.
- Matchings between memory operations at procedure call/return boundaries, where arguments have been saved and later retrieved from the stack.

If we look at the case of load elimination it mostly fits in the first and third cases. A spill-load inside a vector loop might match a spill-store that belongs to its same iteration. A similar kind of storing and reloading of vector data also happens at procedure call/return boundaries. A matching between a load and a store from different iterations of a given loop seems less likely, due to the working set size of scientific vectorizable codes and due to the limited amount of storage space in the store queue. Such matchings could happen, as we shall see, when there are short inter-iteration dependences, where a variable stored in iteration i is later re-loaded in iteration $i+1$.

For store elimination, the second case will be the most frequent one. A spill-store inside a loop will have a match with itself at each different loop iteration since consecutive iterations of a given loop use the same storage locations on the stack to spill registers. However, due to the small size of the VSDQ it is unlikely that much store elimination actually happens. Since spill code typically appears in large loops, and large loops will easily overflow the VSDQ, not much performance improvement is expected from the store elimination scheme.

In section 3.5.1, page 46, we presented three examples of loops having spill code and hinted how the relatively simple FIFO structure of spill code could be used to dynamically eliminate it. There we referred to a "storage space" where spill data would be kept until needed. In the context of the DVA, this storage space is very naturally mapped to the vector load and store queues. Spill stores are held in the VSDQ until they go to memory. If a later spill-load finds a matching in the queue it will copy the data required from the VSDQ directly into the VLDQ.

5.4.2 Load Elimination

The hardware needed to implement the load elimination feature is almost already present in the RDVA architecture and is shown in Fig. 5.16. When a load is found to be identical to a certain store, a bypass pseudo-instruction is sent to a bypass queue. This instruction includes the original slot number in the VSDQ from where data will be copied and the destination slot number in the VLDQ. The bypass queue executes the bypass pseudo-instructions in order by monitoring the original slot in the VSDQ that must be copied. When the slot becomes full, the bypass control starts a copy operation from the VSDQ into the VLDQ. The copying of data can be done in parallel with either a real load or store operation, and thus, in our implementation, the VLDQ is dual ported for writing and the VSDQ is dual ported for reading.

In order to improve the "hit ratio" of loads that find a match in the ASQ, we can try to keep all store data inside the processor even once the data has already been sent to memory. That is, when a store is sent to memory, there is no real need to completely discard it. Instead, a clean/dirty bit can be added to the disambiguation information stored in the store address queue. A slot marked as "clean" will stay in the ASQ as long as there is enough space and will

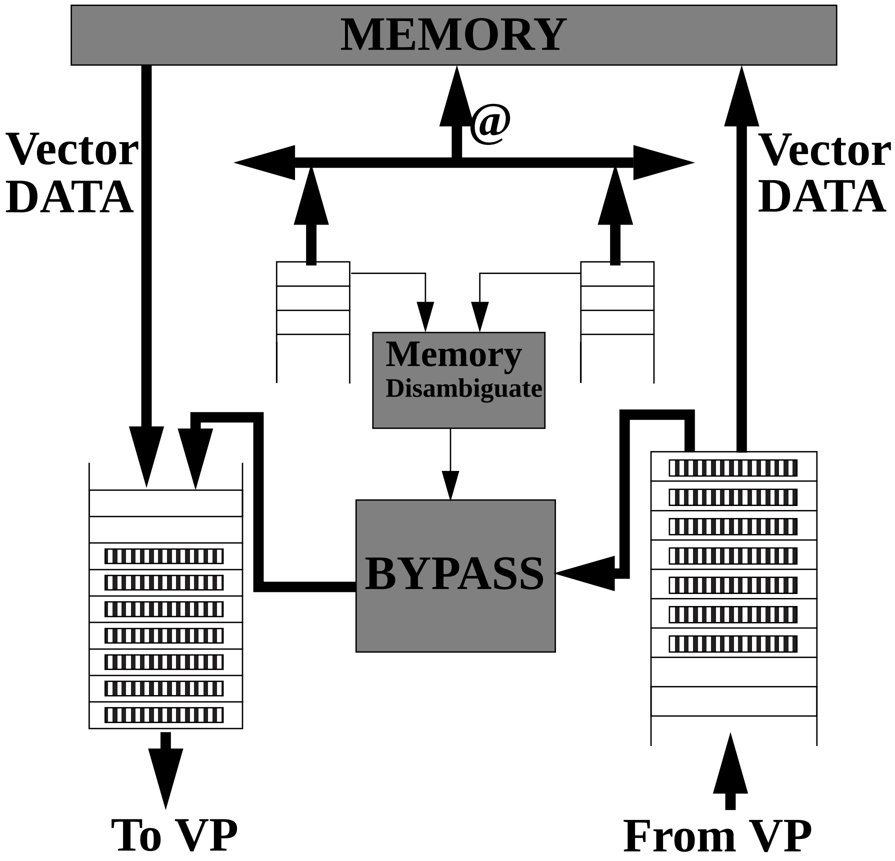

Figure 5.16 Bypassing hardware needed for vector bypasses.

be eligible for bypassing. However, as soon as the AP needs a fresh slot to insert a new store, it will use up any clean slot available. These feature of the dynamic load/store elimination will be refereed to as the “keep clean stores” scheme.

As a final point, note that our implementation does load elimination both for vector and *scalar* data. The reason is that scalar memory operations also go into the ALQ and ASQ and have to be disambiguated against both vector and scalar stores. Therefore, it does not make any sense to disallow scalar load elimination.

5.4.3 Traffic Reduction from Load Elimination

We will first start by looking at the amount of data traffic reduced when load elimination is allowed. Traffic reduction is computed as total number of memory words transferred from/to the memory system in the RDVA architecture divided by total number of memory words transferred from/to the memory system in each configuration under study.

For each experiment with dynamic load/store elimination, we have run simulations increasing the size of the VSDQ from 4 to 128 in powers-of-2 steps. The size of the ASQ, which holds the addresses for the stores was set so that: (a) it was at least 8 slots larger than the size of the VSDQ in order to hold all vector stores plus the scalar stores and (b) it was never smaller than 16 slots (recall that we have 16 slots in the SSDQ, so it makes no sense to have an address store queue smaller than that).

Figure 5.17 presents the traffic reduction ratio achieved over the RDVA machine when load elimination (curve LE in the figure) and load elimination plus "keep clean stores" (curve LE+kcs) are enabled. Note that the X-axis is in log-scale. Data shows that the LE configuration achieves traffic reductions that vary a lot across programs. For the 4 slots VSDQ, we have a first group of programs that have a reduction lower than 1.05 (`hydro2d`, `arc2d`, `nasa7`), a second group of programs that are in the range of 1.05–1.10 comprising `swm256`, `flo52`, `su2cor` and `bdna`, and finally, three programs that are over 1.14 reduction: `tomcatv` (1.14), `trfd` (1.23) and `dyfesm` (1.28). The case of `trfd` and `dyfesm` is especially surprising, since they do not have any spill code at all (see table 3.4, page 45). Clearly, these two programs are experiencing a caching effect. We will discuss them in detail later.

An interesting trend for LE is that it is almost insensitive to the size of the VSDQ. For all programs except `hydro2d` and `tomcatv`, the traffic reduction ratio curve is flat from 4 slots up to 128 slots. This is a clear indicator that bypassing is not as successful as our static analysis might indicate. Either dependencies in the AP or timing issues inside the memory unit are preventing a higher degree of matching of loads and stores.

If we go back to section 3.5.1 and look at the static analysis we did for program `su2cor`, Fig. 3.3 in page 49, we can see that we were expecting some traffic reduction benefits when going from 4 to 8 slots in the VSDQ. Recall from that analysis that with 4 slots in the queue, we should be able to capture 8 spill-loads while having 8 slots in the queue would theoretically allow to capture 14 spill-loads. If this was the case, we should be seeing almost a twofold

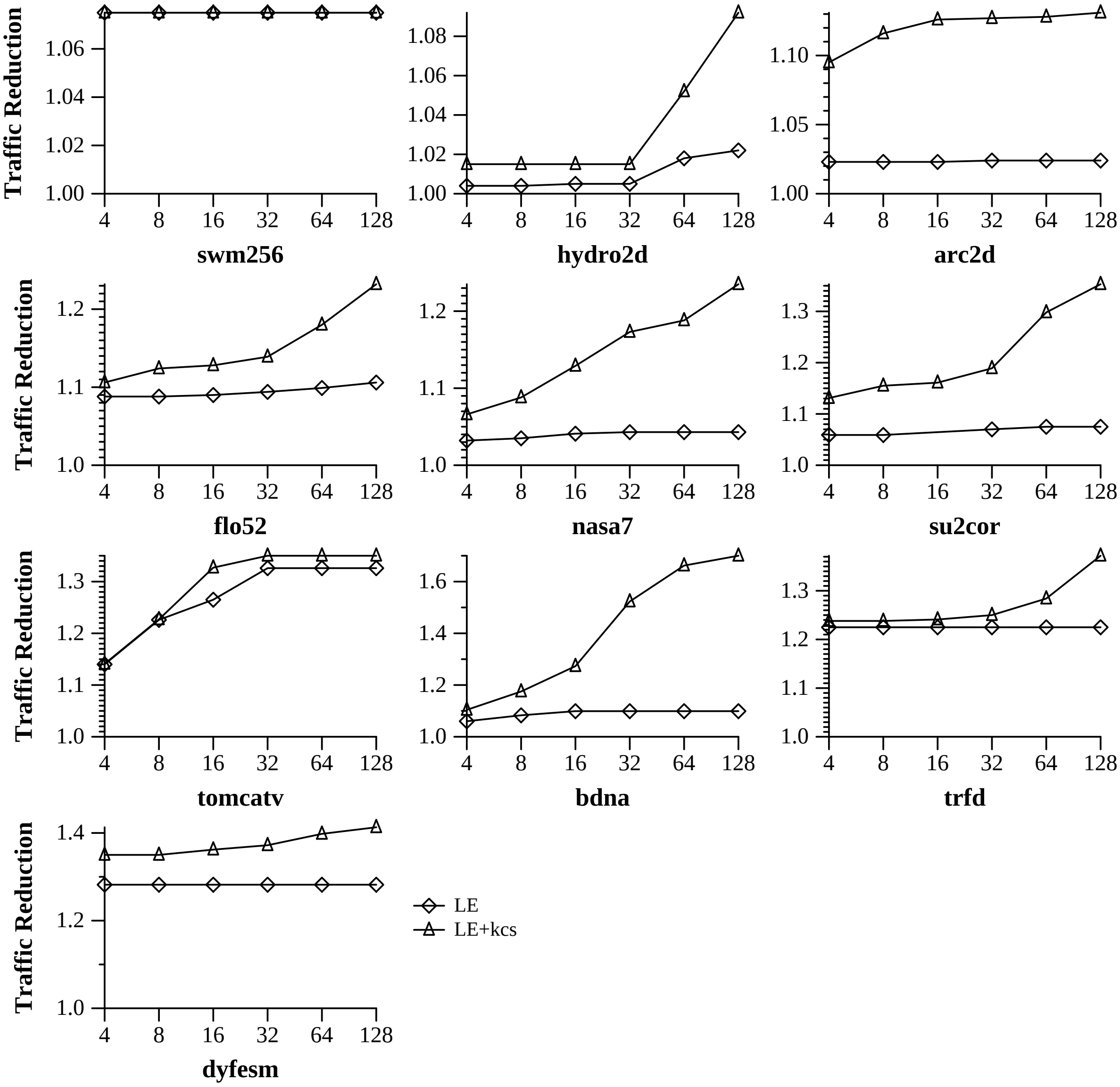

Figure 5.17 Traffic reduction for the LE and LE+kcs models over the RDVA.

increase in traffic reduction when going from 4 to 8 slots and, yet, the curve in Fig. 5.17 is completely flat.

```
0. ld.l 0(a4),v6
1. st.l v1,-5184(fp)
2. mul.d v2,v6,v1
3. add.d v1,v4,v4
4. ld.w -24(fp),a5
5. ld.w -48(fp),a2
6. add.w a1,a5
7. add.w a1,a2
8. ld.w -36(fp),a3
9. add.w a1,a3
10. st.l v4,0(a5)
11. mul.d v0,v7,v1
12. mul.d v3,v6,v4
13. add.d v1,v4,v1
14. ld.l -5184(fp),v4
```

Figure 5.18 Fragment of code from basic block #466 from program `su2cor`. Note that in C34 assembly language, destination operands are on the right and that instructions having two operands (like add #6) have the destination operand as an implicit operand also.

We investigated what happens at run-time for `su2cor`'s block #466 and found that the internal dependencies in the *AP* prevent load/store requests to enter the ALQ/ASQ as fast as desirable. Figure 5.18 shows a fragment of code from basic block #466 to illustrate the point. The two main instructions are store #1 which spills register v1 and load #14 that later reloads the value into register v4. Between them, there is only one other store (number 10), so one would always expect a bypass to occur. The problem is in the dependency encounter by the *AP* between instructions 4 and 6. Consider the sequence of events:

cycle i**:** The *AP* enters load #0 into the ALQ.

cycle $i+1$**:** The memory unit starts execution of load #0 for 128 cycles. At the same time, the *AP* enters store #1 into the ASQ

cycle $i+4$**:** The *AP* enters load #4 into the ALQ.

cycle $i+5$**:** The *AP* enters load #5 into the ALQ.

cycle $i+6$**:** The *AP* blocks on instruction #6 which requires register a5 as an input.

cycle $i + 129$**:** The memory unit executes scalar load #4. Assume for the purposes of the discussion that in the best possible scenario, we have the data being loaded ready at the very next cycle.

cycle $i + 130$**:** The memory unit executes scalar load #5. The *AP* resumes execution and issues add #6.

cycle $i + 131$**:** The memory unit sees the memory port idle during this cycle, and thus decides to send store #1 to memory. At the very same cycle, the *AP* is working on instruction #7, and has not yet reached instruction #14.

What these sequence shows is that even before load #14 gets a chance to be disambiguated against store #1 and a match is found, store #1 will have been sent to memory and discarded. Even if the load entered the queue at cycle $i + 131$, it would be too late since due to race conditions it is not possible to bypass a store that is already in transit to memory. The bottom line is that small scalar dependencies in the *AP* combined with the greediness of the memory unit may produce the effect of stores "escaping" from the queue before they are bypassed.

Keeping clean stores around is a good way of fighting the effect just described in the previous section. Although stores are sent to memory as soon as the memory port is idle, a copy is kept in the queue until the slot is reclaimed. Then, when some cycles later the load enters the queue, it sill finds the data it needs in the VSDQ.

For all programs except `swm256`, configuration *LE+kcs* clearly improves upon plain *LE*, mostly due to the effect just described. For `su2cor`, which was the program under analysis, we can see how the traffic reduction between *LE* and *LE+kcs* at 4 and 8 slots goes, respectively, from 1.06 up to 1.13 and from 1.06 up to 1.16. Moreover, with the inclusion of *kcs* we now start to see the expected benefits that a larger queue size should have, which were not present in the *LE* configuration. Similarly, for most programs we see an increase in traffic reduction when going from 4 to 8 slots under this configuration.

5.4.4 Store Elimination

To perform store elimination extra control hardware is needed. At the very minimum, a kill bit is added to the information stored in the ASQ. In order to avoid race conditions, it is also necessary to mark stores that are actually being sent to memory as "non-bypassable" and to mark stores that some load wants to get its data from as "locked", so that they do not go to memory until all bypass instructions have copied their data.

Now, consider a loop with N store spills and just 1 non-spill store at the end of the loop. Imagine also that the VSDQ is long enough to hold the N+1 stores. After the first iteration of the loop, one would expect that all spill stores are sitting in the queue, and that, as soon as the first store from the second iteration entered the loop, it would match against itself and kill its older copy. In general, this would not be the case due to the fact that the AP always tries to maximize the usage of the memory bus. Thus, if on a given cycle, the bus was free and there was no load instruction ready to be launched, the first store from the queue would have been sent to memory. The first spill store, therefore, would have escaped its "death" and by the time the second iteration was entering the ASQ, some stores might have already been sent to memory.

This situation is far more common that it would seem, and is exacerbated by the fact that load bypassing is very effective. The more loads being bypassed, the more free chimes available for stores to escape their death. This behavior is enforced by the in-order nature of the ASQ. Our early experiments with this simple scheme support this explanation, since they showed only a minimal fraction of store spill traffic being eliminated.

In order to avoid the situation just described and extract some performance gains from store elimination we need to complicate the hardware management of the ASQ with two extra features:

- First, store elimination requires that normal, non-spill stores are allowed to advance previous, older spill stores. This implies out-of-order execution of stores, since some stores will be sent to memory before older stores do so. To support out-of-order execution of stores, the same hardware available for computing load/store dependencies must be replicated in order to compute store/store dependencies.
- Second, the memory unit must treat normal stores with a different policy from spill stores. Recall that we can distinguish the two types of stores by checking if they use register A7 to compute their base address. For normal stores, as soon as they are ready they are eligible for being sent to memory. On the other hand, spill stores need to be held in the queues for as long as possible, and only be sent to memory if the queue overflows. This policy will favor both that (a) more bypasses occur and (b) that the likelihood of a store being eliminated increases, since the longer a certain spill store stays in the queue the

more possible it is for the same store (but belonging to the next iteration of the loop) to enter the queue and replace its older version.

Maintaining spill stores in the queue as long as possible is not a simple task, since it can easily lead to deadlock. Assume, for example, that there is only one store in the whole store queue, and that it is a spill store. Assume also that following this single spill store we have N loads, that the first of these load has a dependency with the store (a real dependency, not an exact match that could be solved through bypassing), and that the ALQ can hold N-1 loads. Under this circumstances, the following are true:

1. The *AP* will block on the Nth load because the ALQ is full,
2. no load is issued due to the dependency of the first load with the store,
3. after some period of time, the *VP* will consume all the data in the VLDQ and will end up blocking waiting for the load data,
4. no more stores will be entered into the ASQ since the *AP* is blocked,
5. the store will never be issued, since it is a spill store and the policy dictates that we should hold it in the queue as long as possible.

Clearly, this is a deadlock situation that must be avoided. There are other situations that can potentially lead to the same type of deadlock. These situations involve any of the output queues of the *VP* and *SP* becoming full. If any of these queues is full, it might be because there are too many store spills in the ASQ and one needs to be sent to memory to prevent deadlock. In summary, the policy regarding spills in the queue is: if there is an overflow condition on queues (2), (6) or (9) (refer to numbers in Fig. 5.2), or there is an overflow in the ASQ or a true dependency has been detected between a load and a spill store, then send a spill store to memory. In all other cases, even if the memory bus is idle, keep spill stores in the queue.

An important drawback of store elimination that must be mentioned is that it does not allow a processor to have precise exceptions. If the processor was interrupted, the software handler would not have a precise state of memory if some store had been eliminated. This characteristic may or may not be a problem depending on the requirements of the machine to which store elimination is added.

5.4.5 Traffic Reduction from Store Elimination

This section will look at the traffic that can be dynamically eliminated when store elimination (SE) is enabled. Nonetheless, we will not study store elimination by itself, since it does not make much sense to try to reduce store traffic without first attempting to eliminate load traffic (which is larger). Therefore we will start looking at how much extra traffic store elimination can reduce when applied on top of the load elimination technique we have already studied.

Figure 5.19 presents the traffic reduction over the RDVA machine when store elimination is enabled. We present three different curves: curve LE is the same as the one shown in Fig. 5.17 and corresponds to doing load elimination only. This curve is plotted for the purposes of comparing the extra traffic reduction achieved from store elimination. Curve LE+SE represents performing both load and store elimination but *without* the out-of-order store feature described in the previous section. This curve is presented to support our point that store elimination requires out-of-order execution of stores and split priorities in order to be useful. Finally, curve LE+SE+ooos plots the traffic reduction for a full-blown version of the dynamic load/store elimination hardware.

The curves confirm the intuition presented in previous sections that, in order to be useful, store elimination requires out-of-order execution of stores and split priorities in the ASQ. As it can bee seen, in all but one program, curve LE+SE+ooos takes off and almost doubles the amount of traffic reduced for queue sizes over 8 slots. Curve LE+SE is almost always side-by-side with the simple LE curve, showing that in most programs it is difficult to take advantage of store elimination alone.

Overall, store elimination can be very useful for large queues. For examples, a queue able to hold 32 slots would cut down memory traffic by a factor in the range of 1.2–1.9. The area cost of such a queue would be 32Kb in our current architecture and has the advantage of a very simple layout. It only requires 2 read ports and 1 write port and has an underlying FIFO structure. If we compare it with a traditional cache, that must include a certain amount of tags and two ports and has a random access pattern, the queue has a clear area advantage. Also, if we compare the queue to a large register file we can see an interesting tradeoff: while adding more registers to the register file and to the ISA level of the machine has very clear advantages, it greatly increases the crossbar logic between the registers and the functional units. The queue has the advantage of providing a way of dynamically reusing data that does not fit within the 8 vector registers while keeping hardware complexity low.

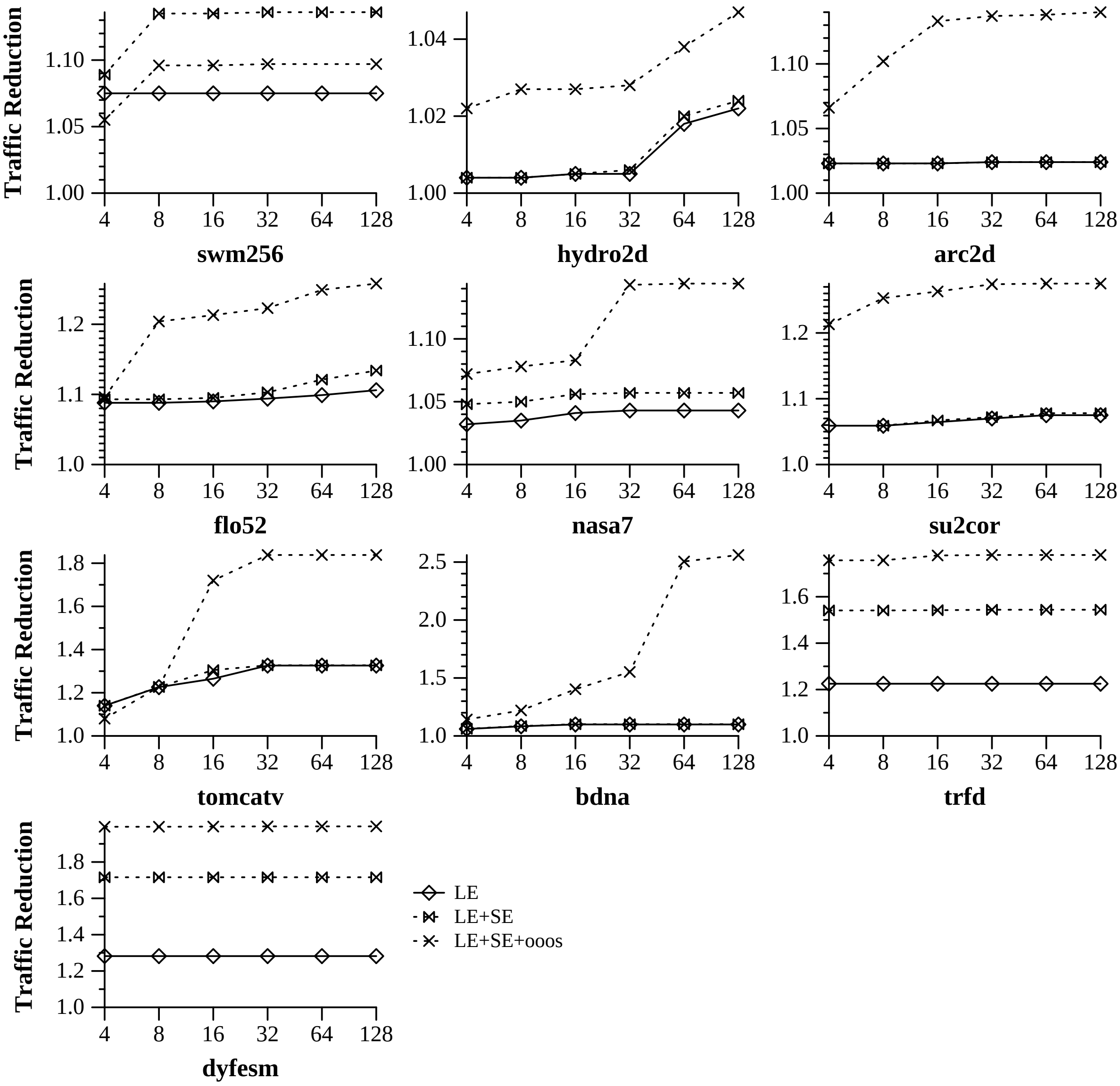

Figure 5.19 Traffic reduction over the RDVA achieved using dynamic load and store elimination.

5.4.6 Traffic Reduction from Keeping Clean Stores

In this section we look at the extra reductions that can be achieved by combining store elimination and the idea of keeping clean stores in the ASQ while there is enough free space.

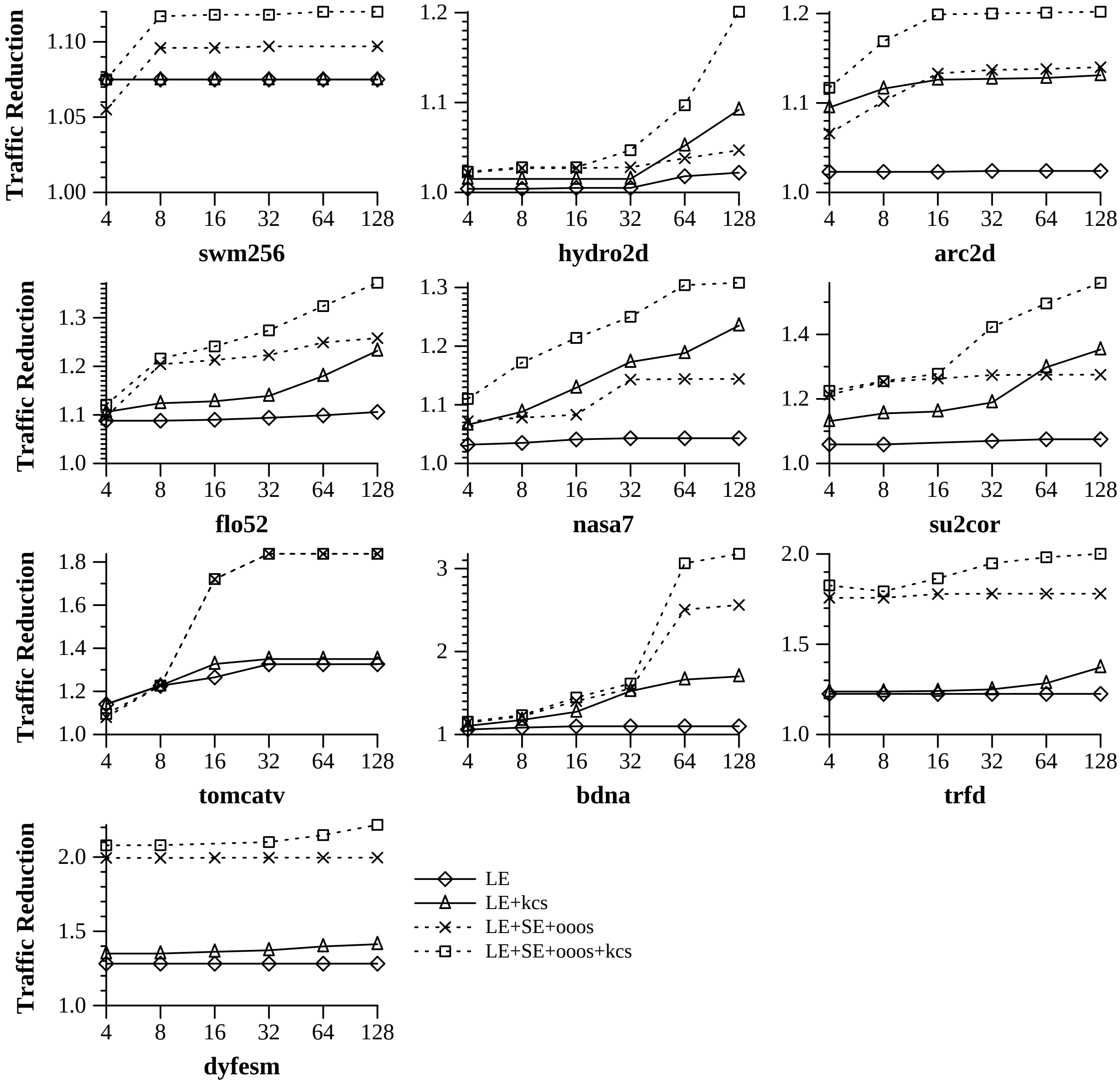

Figure 5.20 Traffic reduction achieved combining load and store elimination and kcs.

Figure 5.20 presents four different curves that highlight the impact of the "keep clean stores" scheme. Curves LE and LE+SE+ooos do not keep clean stores while curves LE+kcs and LE+SE+kcs do include this feature.

We already saw in Fig. 5.17 that the "keep clean stores" scheme did increase the hit ratio for load elimination. Here, in Fig. 5.20 we can see that kcs is also useful combined with store elimination, but typically requires large queues to show significant improvements in traffic reduction. It is interesting to note a couple of cases, programs `nasa7` and `su2cor`, where the less aggressive LE+kcs scheme outperforms the LE+SE+ooos scheme. The explanation is based on the hit ratio of loads in the ASQ: the kcs technique greatly improves the number of loads that bypass their data and this can not be offset by store elimination alone.

5.4.7 A Case Study: `trfd`

Two programs in Fig. 5.17, 5.19 and 5.20 present a very surprising result. In table 3.4, page 45, we saw how both `trfd` and `dyfesm` did not have spill code at all. Yet, for these two programs, load elimination achieves a traffic reduction of over 20% and when store elimination is added the traffic reduction can increase up to a 70% and 90%. Therefore, these two programs are taking advantage of some memory referencing locality other than spill code. We investigated both programs and found that they have short inter iteration dependences that allow matchings of loads and stores from successive iterations. This section will present as a case study the behavior of a loop from `trfd`, which is very similar to what happens with `dyfesm`.

We investigated program `trfd` and found that it spends most of it time in the vectorized DO-LOOP labeled 20 shown in Fig. 5.21. This loop is found in routine OLDA which is called from the main program 7 times. Each time the routine is passed a parameter NUM that goes from 10 to 40 in steps of 5.

By manually propagating variables from the main program into OLDA, we eventually collapsed this code into the code shown in Fig. 5.22. We only show the load/stores relevant to our discussion. Note that loop 20 is fully vectorized, and thus we have removed it from Fig. 5.22. Note also that the notation XRSIQ(MI,MQ) indicates a vector access of length NUM into column MQ of matrix XRSIQ. Analyzing this loop, we can find several points where a load will be eliminated using the bypass path:

- there will be a bypass between store (2) and load (3) whenever MP equals MQ.
- Moreover, when $1 <= MQ < MP-1$ we will have a bypass between store (4) of iteration MQ and load (3) of iteration $MQ+1$.

```
DO 100 MRS=1,NRS
 init code...
 DO 40 MP=1,NP
  DO 30 MQ=1,MP
   MRSPQ=MRSPQ+1
   VAL=XRSPQ(MRSPQ)
   IF(VAL.EQ.ZERO) GO TO 30
   DO 20 MI=1,MORB
    XRSIQ(MI,MQ)=XRSIQ(MI,MQ)+VAL*V(MP,MI)
    XRSIQ(MI,MP)=XRSIQ(MI,MP)+VAL*V(MQ,MI)
   20 CONTINUE
  30 CONTINUE
 40 CONTINUE
 update code deleted....
100 CONTINUE
```

Figure 5.21 Loop corresponding to basic block #200 from `trfd`.

```
DO 100 MRS=1,(NUM*(NUM+1))/2
 init code...
 DO 40 MP=1,NUM
  DO 30 MQ=1,MP
   (1) Vector-load XRSIQ(MI,MQ)
   (2) Vector-store XRSIQ(MI,MQ)
   (3) Vector-load XRSIQ(MI,MP)
   (4) Vector-store XRSIQ(MI,MP)
  30 CONTINUE
 40 CONTINUE
 update code deleted....
100 CONTINUE
```

Figure 5.22 Loop corresponding to basic block #200 from `trfd`, vectorized and with variables propagated from the main program.

- There is also a potential bypass between store (2) of iteration $\langle MP, MQ\rangle = \langle j-1, i\rangle$ and load (1) of iteration $\langle MP, MQ\rangle = \langle j, i\rangle$. For this latter bypass to be realized, the VSDQ must be at least $(MP-1)\times 2$ slots deep, in order to hold all the stores in between.

A very similar analysis for store killing will lead to the conclusion that:

- whenever $MQ > 1$ there is a store kill between store (4) of iteration $\langle *, i-1\rangle$ and store (4) of iteration $\langle *, i\rangle$.
- whenever the queue is larger than $(MP-1)\times 2$ slots, there might be a store kill between store (2) of iteration $\langle j-1, i\rangle$ and store (2) of iteration $\langle j, i\rangle$.

This example shows how load and store elimination can sometimes also be useful as a caching mechanism for normal program variables and not only spill code.

5.4.8 Speedup

We now turn to the question of what execution time improvements are derived from all the traffic reductions we have seen so far. Figure 5.23 presents the speedups obtained for the best performing elimination schemes over the plain RDVA machine with no dynamic elimination at all. We have selected for the purposes of comparison the same curves as in Fig. 5.20: load elimination alone (LE), load elimination with keep clean stores (LE+kcs), load and store elimination with out-of-order execution of stores (LE+SE+ooos) and load and store elimination with out-of-order store execution plus the keep clean stores scheme (LE+SE+ooos+kcs).

How is it possible that such high reductions in total memory traffic do not translate into bigger speedups ? To understand this fact, consider that from the point of view of the *VP*, neither load bypassing nor load killing are reducing the amount of data movement it has to perform. That is, inside the *VP*, all `qmov`'s corresponding to spill loads/stores are still being executed, whether they are killed or not. Thus, all this reduction of traffic is only seen outside the *AP* and, in general, would be mostly useful in multiprocessing systems. Needless to say, any reduction on the load of the memory system usually translates into higher memory performance, but since decoupling can cover up long memory latencies, these benefits are not seen in our experiments.

It is interesting to see the negative effect that out-of-order execution of stores can have in execution time in `bdna`. What happens in this program is that forcing spill stores to stay in the queue for as long as possible hurts the critical path of the program. Spill-stores that

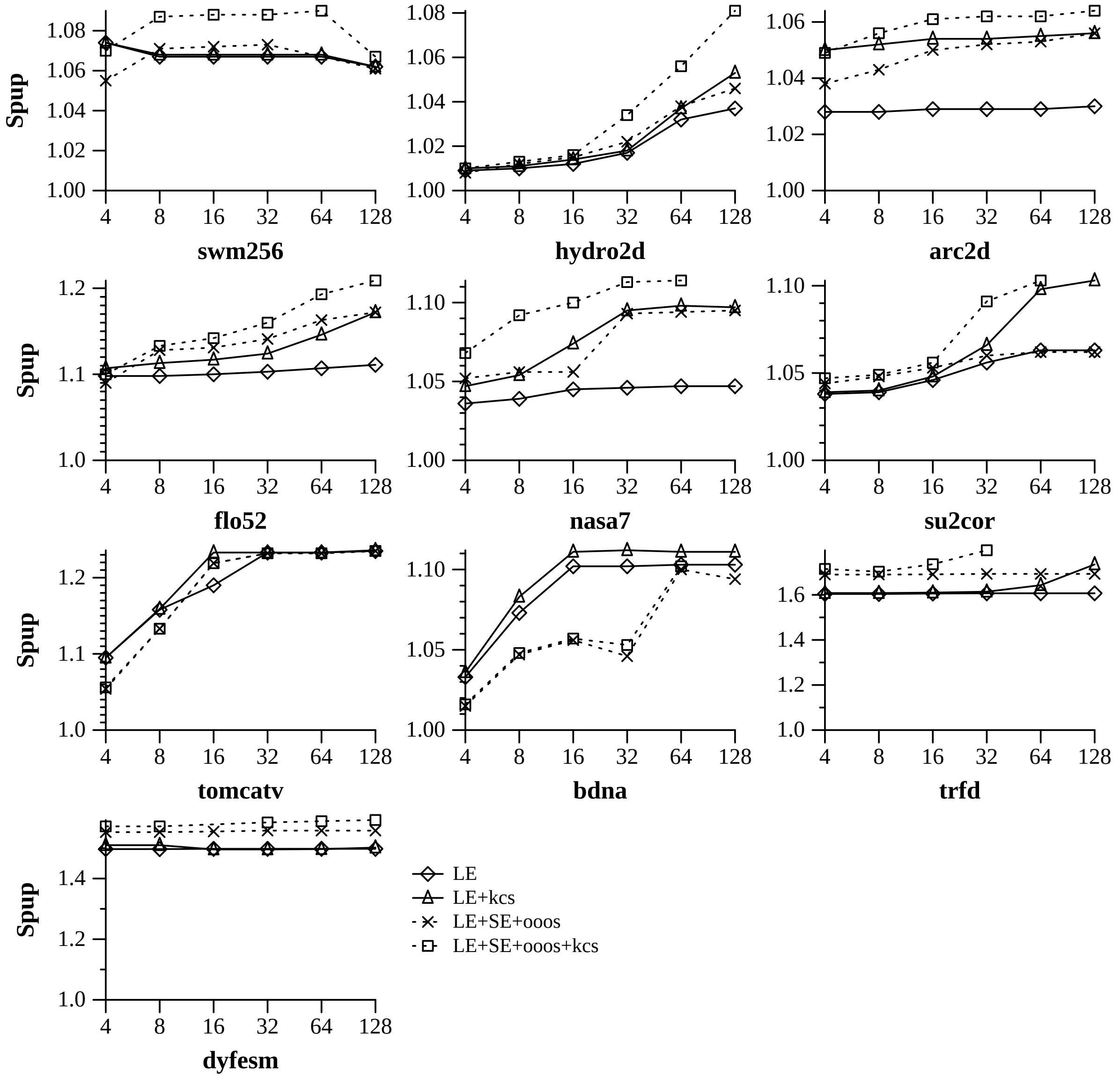

Figure 5.23 Performance of several dynamic load/store elimination schemes over the RDVA.

could have been sent to memory in otherwise idle memory chimes were kept in the queue and later, when the queue was full, these stores took up critical cycles on the memory port.

Nonetheless, the overall performance of dynamic load store elimination is quite good: speedups for a moderately-sized queue of 16 slots range anywhere 1.05 and 1.10 for most programs and go up to a 1.15 for `flo52`, a 1.55 for `dyfesm` and a 1.7 for `trfd`.

5.5 RELATED WORK

There has been a lot of research and development in the area of decoupled architectures, but has mostly focused on scalar architectures. To the best of our knowledge, this is the first attempt at using decoupling in a vector architecture. Part of the work that has been presented here was also presented in [EV96]. The closest attempt at putting together the concept of a vector architecture and a decoupled architecture was presented by Young [You85]. That work attempted at using the regularity of the queues found in a DAE to achieve pseudo-vector processing. Nonetheless, the architecture under study did not have a true vector instruction set such as the one being studied here.

Decoupling was first formally introduced by Smith [Smi82b], although some previous machines, such as the MAP-200 machine described in Cohler and Storer [CS81], did already show a decoupled architecture. The MAP-200 had a processing unit, the APU, and a separate addressing unit, the APS. A distinctive characteristic of this machine was that each unit had a separate instruction memory for each processor.

The decoupled access/execute machine introduced by Smith [Smi82b] and further described in [Smi84] was loosely based on the Cray-1 instruction set and also had separate units for memory accessing and program computations. This initial design studied both the possibility of having a single program counter and an instruction splitter and having two separate program counters. From this initial design evolved the PIPE architecture, described in [SPKG83, GHL+85, You88, FP91a].

The synchronous distributed processor (SDP) described by Shively [Shi82] was a special purpose digital signal processor that also presented decoupling between the memory accessing unit and the computation unit. Although a special purpose machine, it featured input and output operand queues for the arithmetic unit.

The Structured Memory Access architecture (SMA) [SD84] was a decoupled architecture with two units communicating through queues. The access unit of the SMA was a more complex processor than typically found in DAE's having special features to support efficient access to data structures.

The Fortran Oriented Machine (FOM) [BW83] was an IBM research project that introduced distinct execute and memory accessing units tied together by an "exchange/convert unit" that served the role of a buffering unit that held data in copy queues.

The Astronautics ZS-1 was a commercial attempt lead by J. E. Smith at producing a decoupled access execute computer [SDV$^+$87, Smi89]. The machine had two separate pipelines, the A and X pipelines that performed memory accessing and floating point computations, respectively. Both pipelines were fed by a single instruction fetch unit that split the incoming non-decoupled instruction stream. The machine is further studied by Mangione-Smith et al. [MSAD90], where they show the latency tolerance properties of this particular DAE design. The ZS-1 was also compared to an IBM RS/6000 in [MSAD91b].

The Wulf Machine (WM) introduced by Wulf [Wul92] represents an hybrid model between a decoupled machine and some aspects of vector processors. The main characteristic of the WM is that it has streaming memory operations that very closely resemble vector loads and stores. This stream operations fill (or drain) a set of queues as in a DAE and then normal scalar instructions consume data from the queues in a RISC-style.

Love and Jordan [LJ90] looked at the performance differences of dynamic scheduling versus static scheduling, by comparing a DAE to a VLIW [CDN92, CNO$^+$87, KG92, MCmWH$^+$92, SZTH92, WS91] processor. Their conclusion is that the two architectures performed equally well for typical workloads while the implementation complexity would be different.

The latency tolerance properties of decoupled architectures have been studied at length by several authors [CHM89, HC91]. Kurian et al. [KHC92, KHC94] presented simulations that closely match some of the results seen in section 5.3.7, showing that the decoupled version of an architecture could tolerate much larger memory latencies than its non-decoupled counterpart. We note that, to the best of our knowledge, previous studies on memory latency effects had never contemplated such large memory latencies (in processor cycles) as we do in this work. Kurian and colleagues, for example, look at memory latencies of 5, 10 and 15 cycles and compare the performance of a traditional machine, a standalone DAE and a DAE enhanced with a data cache. They conclude that for latencies in the range of 15 cycles (*much* lower than the ones used here) the cache can be of help to tolerate and alleviate the memory bandwidth problem. Our work shows how the combination of decoupling and vector memory accessing can yield even better results, as we have seen memory latencies of 100 cycles being successfully tolerated. Windheiser and Jalby [WJ91] provide an analytical framework to study the performance of decoupled machines and Young and Goodman presented simulation results in support of architectural data queues [YG84].

Bird et al. [BRT93] also assessed the latency tolerance properties of decoupled access execute architectures. In their work they also introduce control decoupling, a technique whereby control instructions, such as branches, jumps and procedure calls are further decoupled from the access and execute processor. Their technique allows to alleviate some branching penalties that might occur in traditional DAE's. Recently, a failed commercial attempt at producing a decoupled architecture yielded the ACRI-1 architecture [TM95]. This architecture not only had access/execute decoupling but also introduced control decoupling as described in [BRT93].

Dorojevets and Oklobdzija [DO95] introduce an architecture where the decoupling and multithreading concepts were merged to exploit as much instruction level parallelism as possible.

5.6 SUMMARY

In this chapter we have described a basic decoupled vector architecture (DVA) that uses the principles of decoupling to hide most of the memory latency seen by vector processors.

The DVA architecture shows a clear speedup over the REF architecture even when memory latency is just 1 cycle. This speedup is due to the fact that the *AP* slips ahead of the *VP* and loads data in advance, so that when the *VP* needs its input operand they are (almost) always ready in the queues. Even if there is no latency in the memory system, this "slipping" produces a similar effect as a prefetching technique, with the advantage that the *AP* knows which data has to be loaded (no incorrect prefetches). Thus, the partitioning of the program into separate tasks helps in exploiting more parallelism between the *AP* and *VP* and translates into an increase in performance, even in the absence of memory latency. Moreover, as we increase latency, we see how the slopes of the curves of the execution time of the benchmarks remain fairly stable, whereas the REF architecture is much more sensitive to the increase in memory latency.

Finally, we have seen that this speed improvements can be implemented with a reasonable cost/performance tradeoff. Section 5.3.4 has shown how the length of the queues does not need to be very large to allow for the decoupling to take place. A vector load queue of four slots is enough to achieve a high fraction of the maximum performance obtainable by an infinite queue. On the other side, the vector store queue does not need to be very large. Our experiments varying the store queue length indicate that a store queue of 2 elements achieves almost the same performance as one with sixteen slots.

The dynamic scheduling has proved useful to hide memory latencies and improve performance. Therefore, a logical step would be to extend this dynamic issue mechanism to all parts of the architecture, thereby allowing all types of instructions to execute out of order. This will be studied in the following chapter, where using register renaming all types of instructions are executed in a greedy way and only true dependences are preserved.

Regarding spill code, we have presented a combination of techniques that use the data stored in the VSDQ to detect and eliminate redundant loads. The elimination of loads is done by physically copying data from the VSDQ back into the VLDQ. We have also presented a store elimination technique that detects repeated stores to the same memory locations and eliminates to oldest store. These techniques achieve two goals: memory traffic reduction and program speedup. Looking at the best performing combination of techniques, LE+SE+ooos+kcs, we see that it reduces memory traffic by factors of 1.1 up to 2.0, and provides associated speedups in the range 1.05–1.7.

6

OUT-OF-ORDER VECTOR ARCHITECTURE

Summary

In this chapter we apply out-of-order execution and register renaming to our baseline reference architecture. We will show that these dynamic instruction scheduling techniques can be used to significant advantage in vector processors, in the same way that they are useful for superscalar processors. We will present performance data varying two key parameters: number of physical registers used to implement renaming and memory latency. We will look at how renaming can provide an easy way to implement precise exceptions – generally a difficult task in vector machines. We will also show that when memory latencies are varied, using out-of-order techniques allows much latency tolerance than traditional vector implementations. Finally, a new technique based on register renaming is targeted at dynamically eliminating spill code.

6.1 INTRODUCTION

The previous chapter has shown that dynamic instruction scheduling in the form of decoupling can be very useful to improve vector performance and mask memory latency. Yet, the approach taken was rather conservative, in the sense that the dynamic instruction reordering only happened in two ways: computation and memory accessing instructions executed out-of-order with respect to each other, and load and store instructions also executed out-of-order with respect to each other.

The next logical step from this limited form of dynamic scheduling would be to see if allowing more out-of-order execution among all types of instructions could be of any help. In particular, reordering among computation instructions could help in better using the two vector functional units. Reordering among all memory instructions could also improve the memory port occupation.

Out-of-order execution has long been used as a latency tolerance technique and as a way to increase instruction level parallelism. It was present in the IBM 360/91 [AST67], which had a set of intermediate buffers that decoupled the core cpu from the memory access and used a renaming mechanism by Tomasulo [Tom67] to allow operations execute fully out-of-order. Register renaming and out-of-order instruction issue [Joh91, SS95] are now commonly used in superscalar processors [DA92, BDHS94, MBG95, Hun95, CS95, WPS95, DNS95, SEI+95, Pap96, PW96, Yag96]. Out-of-order execution improves performance by dynamically reordering instructions of a program and executing them in a greedy way based on operand availability and data dependences. The goal of out-of-order execution is to detect instructions that do not have dependences and execute them in parallel. In order to do so, an out-of-order engine must fetch several instructions at once and compare all their data requirements to determine a partial execution ordering that guarantees a semantically correct execution and maximizes processor usage. This data dependence detection must be made at two semantically different levels: on one hand, register names implicitly define dependences among instructions. On the other hand, memory instructions must be carefully disambiguated to reveal potential data conflicts.

In superscalar processors, out-of-order execution was initially introduced without register renaming. Most of the first generation of superscalar processors had an expensive decoder that tracked all types of dependencies between a pair of instructions (true dependences, anti-dependences and output dependences) [Hsu94, AAD+93, TGN95]. Later, register renaming has gained a widespread acceptance as being a powerful technique to eliminate many artificial dependencies (anti- and output dependences) and is now incorporated in most high performance out-of-order designs, such as the PA-8000 [Hun95], the MIPS R10000 [Yag96] or the ALPHA 21264 [Kel96].

These techniques have not been used to date in vector processors for reasons that are not very clear. On one side, it is true that the semantics of a vector instruction contain a high degree of instruction level parallelism, since a single instruction can potentially specify many operations. The compiler in vector machines is left with the burden of discovering and extracting instruction level parallelism through vectorization techniques. Nonetheless, it is also true that inter-instruction dependences can make a vector processor stall the same way as superscalar processor. As we have seen in chapter 4, each time a stall stops the instruction decoder the probability that the memory port goes idle is high, and this results in a loss of performance and latency tolerance.

In the initial times of vector architectures, the ECL costs of implementing a complex out-of-order dispatching mechanisms clearly out-weighted the perceived advantage. In today's CMOS technology it is much simpler and will undoubtedly appear in most vector processor designs. Another potential reason for not introducing out-of-order execution is that precise interrupts have always been difficult in the vector environment. Since vector processors have many operations in-flight at any given point in time, restoring the processor to a precise state after an interrupt is generally recognized as a difficult task [SHH90]. Adding to this complexity the task of managing out-of-order execution was generally regarded as excessive. As an example of the complexity of precise interrupts in vector machines, consider that most vector mainframes do not support virtual memory in the fully-flexible way that superscalar workstations do [IW91]. Mostly, this is due to the complexity of taking a page fault in a precise manner and restoring the processor state after operating system intervention.

Despite all these potential reasons, introducing out-of-order execution and register renaming in a vector architecture can have many advantages. First, as already mentioned, a performance advantage due to better resource utilization. Second, derived from this better resource utilization, memory bandwidth is fully used and the same latency tolerance properties as in the decoupling technique can result. Third, register renaming can be of great help to maintain a precise, coherent view of the processor state at any given point in time. Since register renaming is able to distinguish between the "architectural state" and the "speculative state" one can always revert to the architectural state in face of an exception or a page fault. This precise interruption capability can lead to easy implementations of virtual memory.

Moreover, all these potential advantages of out-of-order execution and register renaming can be harnessed at a much lower cost than in superscalar architectures. It is widely recognized that a major problem in the performance of superscalar architectures is the bottleneck created at the fetch and issue state by the wide and complex instruction fetchers and decoders. There have been many studies on the available instruction level parallelism in scalar programs [JW89, Jou89, Wal91, SJH89] that conclude that there is a limit to the amount of exploitable instruction parallelism due to branches, hazards and cache misses. The advantage of trying to exploit some degree of out-of-order execution in a vector processor is that the compiler has taken the burden of finding a lot of available parallelism and has exposed it through the semantically-rich vector instructions. Thus a vector processor need not have parallel fetching and issue to extract full performance of a machine. With a simple single issue mechanism a vector machine can keep its functional units busy for many cycles.

This chapter will look at the performance that can be achieved through the use of dynamic out-of-order issue and register renaming. We will show that by using out-of-order issue and register renaming techniques in a vector processor, performance can be improved greatly. Dynamic instruction scheduling allows memory latencies to be overlapped more completely – and uses the valuable memory resource more efficiently in the process. Moreover, once renaming has been introduced into the architecture, it enables straightforward implementations of precise exceptions, which in turn provide an easy way of introducing virtual memory, without much extra hardware and without incurring a great performance penalty.

For the spill code problem that we have been studying, we also present a new technique aimed at dynamically eliminating redundant loads. Using this technique, memory traffic can be significantly reduced and performance further increased. This technique uses the renaming tables present in the architecture to execute some loads in almost zero cycles and without accessing main memory. The result is a decrease in total memory traffic and a performance improvement.

6.2 THE DYNAMIC OUT-OF-ORDER VECTOR ARCHITECTURE (OOOVA)

The out-of-order and renaming version of the reference architecture, OOOVA, is shown in Fig. 6.1. It is derived from the reference architecture by applying a renaming technique very similar to that found in the R10000 [Yag96]. Instructions flow in-order through the Fetch

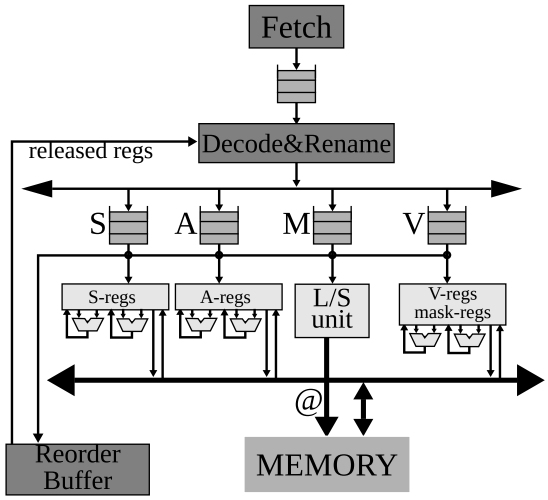

Figure 6.1 The Out-of-order and renaming version of the reference vector architecture.

and Decode/Rename stages and then go to one of the four queues present in the architecture based on instruction type. At the rename stage, a mapping table translates each virtual register into a physical register. There are 4 independent mapping tables, one for each type of registers: A, S, V and mask registers. Each mapping table has its own associated list of free registers. When instructions are accepted into the decode stage, a slot in the reorder buffer is also allocated. Instructions enter and exit the reorder buffer in strict program order. When an instruction defines a new logical register, a physical register is taken from the free list, the mapping table entry for the logical register is updated with the new physical register number and the old mapping is stored in the reorder buffer slot allocated to the instruction. When the instruction commits, the old physical register is returned to its free list. Note that the reorder buffer does *not* hold data, only register names. Data is always kept in the vector register file and thus, this reorder buffer structure is very small.

Apart from the four types of registers already mentioned, the reference vector machine has two important registers that have to be taken care of: the vector length (VL) and vector stride (VS) registers. These two registers control the execution of all vector instructions. We decided to handle these two registers in the same way as the A registers, and to rename and re-map them along with A registers. Renaming is clearly necessary because vector instructions are executing out of order and hence need to be able to identify their own VL and VS. Maintaining them along with the A register file is natural because reads and writes to VL and VS are

Register File	Logical regs.	Physical regs.	Read ports	Write ports
A-regs	8+2+3	64	9	4
S-regs	8+1	64	7	4
mask-regs	1	8	3	2
V-regs	8	9–64	1	1

Table 6.1 Register files characteristics. The extra registers added to the A-regs are the VL and VS registers and 3 condition code registers. The extra register added to the S-regs is a condition code register. For the V-regs, the read/write ports are per physical vector register.

always done from/to A-registers and using A-type instructions. The V register read/write ports have been modified from the original scheme and in the OOOVA each vector register has 1 dedicated read port and 1 dedicated write port. The original banking scheme of the register file can not be kept since renaming shuffles all the carefully scheduled read/write ports and, therefore, would induce a lot of port conflicts. Table 6.1 summarizes all register files present in the OOOVA.

Main Pipelines

There are four main pipelines in the OOOVA architecture. Figure 6.2 shows these four pipelines. Instructions flow through the Fetch and Decode/Rename stages and then go to one of the four queues present in the architecture based on instruction type. Unless otherwise noted, the fetch stage, the decode stages and all four queues only process a maximum of 1 instruction per cycle. Committing instructions proceeds at a faster rate, and up to 4 instructions may commit per cycle.

The A, S and V queues monitor the ready status of all instructions held in the queue slots and as soon as an instruction is ready, it is sent to the appropriate functional unit for execution. Processing of instructions in the M queue proceeds in two phases. First, instructions proceed in-order through a 3 stage pipeline comprising the Issue/Rf stage, the Range stage and the Dependence stage. After they have completed these three steps, memory instructions can proceed out of order based on dependence information computed and operand availability (for stores). During the Issue/Rf stage, a memory operation reads the A-register needed to compute its base memory address, and also reads the VL and VS registers if necessary.

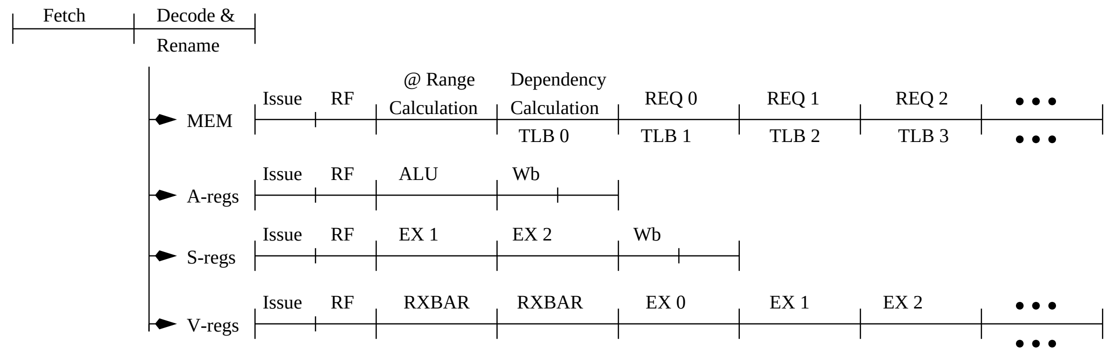

Figure 6.2 The Out-of-order and renaming main instruction pipelines.

At the Range stage, the range of all addresses potentially modified by a memory instruction is computed, the same way as we did in the DVA (see section 5.2, page 84). This range is used in the following stage for run-time memory disambiguation. The range is defined as all bytes falling between the base address (called Range Start) and the address defined as $baseaddress + (VL - 1) * VS$ (called Range End). Note that the multiplier can be simplified because $VL - 1$ is short (never more than 128 in the reference architecture), and the product $(VL-1)*VS$ can be kept in a non-architected register and implicitly updated when either VL or VS is modified. In the Dependence stage, using the Range Start/Range End addresses, the memory instruction is compared against all previous instructions found in the queue. Once a memory instruction is free of any dependences, it can proceed to issue memory requests.

Machine Parameters

All instruction queues are set at 16 slots. The reorder buffer can hold 64 instructions. The machine has a 64 entry BTB, where each entry has a 2-bit saturating counter for predicting the outcome of branches. Also, an 8-deep return stack is used to predict call/return sequences. Both scalar register files (A and S) have 64 physical registers each. The mask register file has 8 physical registers.

Commit Strategy

For V registers we start with an aggressive implementation where physical registers are released at the time the vector instruction *begins* execution. For example, consider the vector instruction: `add v0,v1-->v3`. At the rename stage, v3 will be re-mapped to, say, physical vector register 9 (ph9), and the old mapping of v3, which was, say, physical vector register 12 (ph12), will be stored in the reorder buffer slot associated with the `add` instruction. When the `add` instruction begins execution, we mark the associated reorder buffer slot as ready to be committed. When the slot reaches the head of the buffer, ph12 is released. Note that, due to the semantics of a vector register, when ph12 is released, it is guaranteed that all instructions needing ph12 have begun execution at least one cycle before. Thus, the first element of ph12 is already flowing through the register file read crossbar. Even if ph12 is immediately reassigned to a new logical register and some other instruction starts writing into ph12, the instructions reading ph12 are at the very least one cycle ahead and will always read the correct values. This type of releasing does not allow for precise exceptions, though. Section 6.4 will change the release algorithm to allow for precise exceptions.

Decoupling vs. Out-of-Order execution

At this point it would be good to emphasize the differences between the decoupling technique presented in the last chapter and out-of-order execution.

The main difference is in the amount of instruction reordering allowed in the OOOVA with respect to the DVA. Figure 6.3 tabulates the degree of dynamic reordering in each architecture. For each architecture type we have a table that splits all instructions into three groups: loads, stores and computations. Each cell in the table indicates whether two instruction classes execute out-of-order with respect each other or not. For example, in a DVA, loads execute out-of-order with respect to stores and thus we have marks in cells (1,2) and (2,1) of Fig. 6.3 (a). The table shows that out-of-order execution is more aggressive than decoupling by allowing all types of instruction to execute out-of-order with respect each other.

A second difference between out-of-order execution and decoupling relates to the way in which data fetched ahead of time is stored. Both types of architectures improve upon in-order issue by pre-advancing instructions. In particular, executing memory instruction ahead of time is the key to achieve latency tolerance. In the DVA, these memory loads executed ahead of time keep their data in a regular and simple queue structure. This queue, the VLDQ in Fig. 5.2, page 83, has both a low fan-out and fan-in: two read and two write ports were enough to achieve a high memory throughput. On the other hand, the OOOVA machine uses the central vector register file to store its pre-advanced data. This central file is multi-ported because it has to feed two functional units plus the memory port (5 read ports and 3 write ports

	Load	Store	Comp.
Load		✓	✓
Store	✓		✓
Comp.	✓	✓	

a) DVA

	Load	Store	Comp.
Load	✓	✓	✓
Store	✓	✓	✓
Comp.	✓	✓	✓

b) OOO

Figure 6.3 A comparison of the DVA and out-of-order execution. Each cell in the table indicates whether two instruction classes execute out-of-order with respect each other or not. A mark in the cell indicates that the two instruction groups *do* execute out-of-order.

minimum). Thus, enlarging the register file might have a negative impact on the crossbar complexity connecting vector registers and functional units.

6.3 PERFORMANCE RESULTS

6.3.1 Performance of the OOOVA

In this section we present the performance of the OOOVA and compare it with the reference architecture. We consider both overall performance in speedup and memory port occupation.

Speedup

The effects of adding out-of-order execution and renaming to the reference architecture can be seen in Fig. 6.4. For each program we plot the speedup over the reference architecture when the number of physical vector registers is varied from 9 to 64 (memory latency is set at 50 cycles). In each graph, we show the speedup for two OOOVA implementations: "OOOVA-16" has length 16 instruction queues, and "OOOVA-128" has length 128 queues. We also show the maximum ideal speedup that can theoretically be achieved ("IDEAL", along the

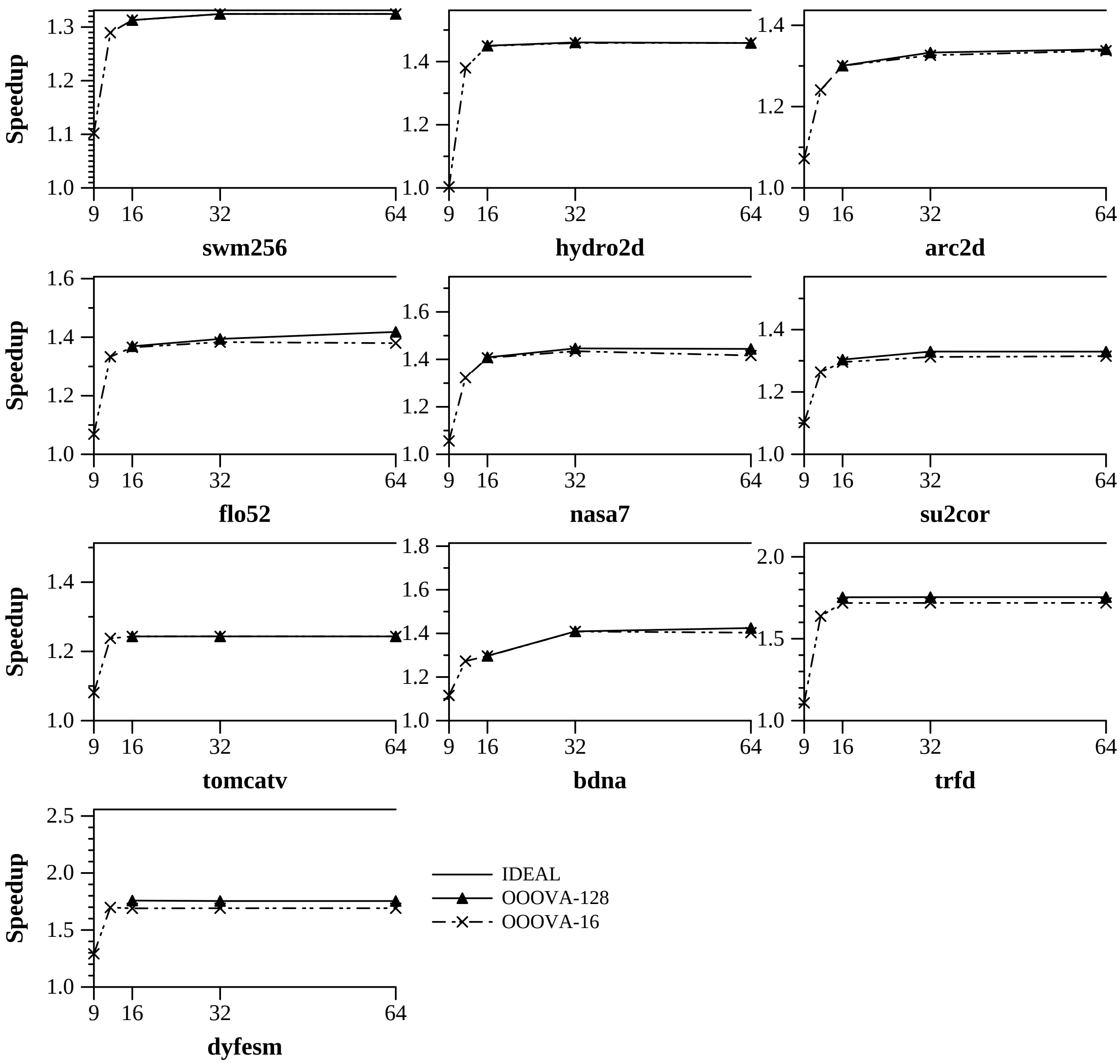

Figure 6.4 Speedup of the OOOVA over the REF architecture for different numbers of vector physical registers.

top of each graph), computed as we already did in section 4.2, page 58, and in section 3.6 (remainder: we use the total number of cycles consumed by the most heavily used vector unit (FU1, FU2, or MEM), essentially eliminating all data and memory dependences from the program, and considering performance limited only by the most saturated resource across the entire execution).

As can be seen from Fig. 6.4, the OOOVA significantly increases performance over the reference machine. With 16 physical registers, the lowest speedup is 1.24 (for `tomcatv`). The highest speedups are for `trfd` and `dyfesm` (1.72 and 1.70 resp.); the remaining programs give speedups of 1.3–1.45. For numbers of physical registers greater than 16, additional speedups are generally small. The largest speedup from going to 64 physical registers is for `bdna` where the additional improvement is 8.3%. The improvement in `bdna` is due to an extremely large main loop, which generates a sequence of basic blocks with more than 800 vector instructions. More physical registers allow it to better match the large available ILP in these basic blocks. On the other hand, if the number of physical vector registers is a major concern, we observe that 12 physical registers still give speedups of 1.63 and 1.70 for `trfd` and `dyfesm` and that the other programs are in the range of 1.23 to 1.38. These results suggest that a physical vector register with as few as 12 registers is sufficient in most cases. A file with 16 registers is enough to sustain high performance in every case.

When we increase the depth of the instruction queues to 128, the performance improvement is quite small (curve "OOOVA-128"). Analysis of the programs shows that two factors combine to prevent further improvements when increasing the number of issue queue slots. First, the spill code present in large basic blocks induces a lot of memory conflicts in the memory queue. These conflicts typically block the queue and result in idle periods on the memory port. Second, the lack of scalar registers sometimes prevents the dynamic unrolling of enough iterations of a vector loop to make full usage of the memory port.

Memory Port Usage

The out-of-order issue feature allows memory access instructions to slip ahead of computation instructions, resulting in a compaction of memory access operations. The presence of fewer wasted memory cycles is shown in Fig. 6.5. This figure contains the number of cycles where the address port is idle divided by the total number of execution cycles. Bars for the reference machine, REF, and for the out-of-order machine, OOOVA, are shown. The OOOVA machines has 16 physical vector registers and a memory latency of 50 cycles. With OOOVA, the fraction of idle memory cycles is more than cut in half in most cases. For all but two of the benchmarks, the memory port is idle less than 20% of the time.

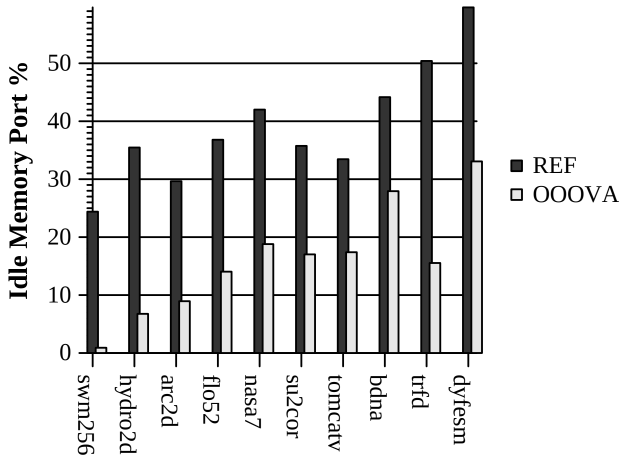

Figure 6.5 Percentage of Idle cycles in the memory port for the Reference architecture and the OOOVA architecture. Memory latency is 50 cycles and the vector register file holds 16 physical vector registers.

Resource Usage

We now consider resource usage for the OOOVA machine and compare it with the reference machine. This is illustrated in Fig. 6.6. The same notation as in Fig. 4.3, page 61, is used for representing the execution state. As in the previous subsections, the OOOVA machine has 16 physical vector registers and memory latency is set at 50 cycles.

Figure 6.6 shows that the major improvement is in state $\langle\ ,\ ,\ \rangle$, which has almost disappeared. Also, the fully-utilized state, $\langle FU2, FU1, LD\rangle$, is relatively more frequent due to the benefits of out-of-order execution. As we have already seen, the availability of more than one memory instruction ready to be launched in the memory queues allows for much higher usage of the memory port. Vector loads are typically executed ahead of time and their associated data arrives at a register before it is required. This slip-ahead effect helps hiding memory latency and improves overall performance.

6.3.2 Tolerance of Memory Latencies

One way of looking at the advantage of out-of-order execution and register renaming is that it allows long memory latencies to be hidden. In previous subsections we showed the benefits

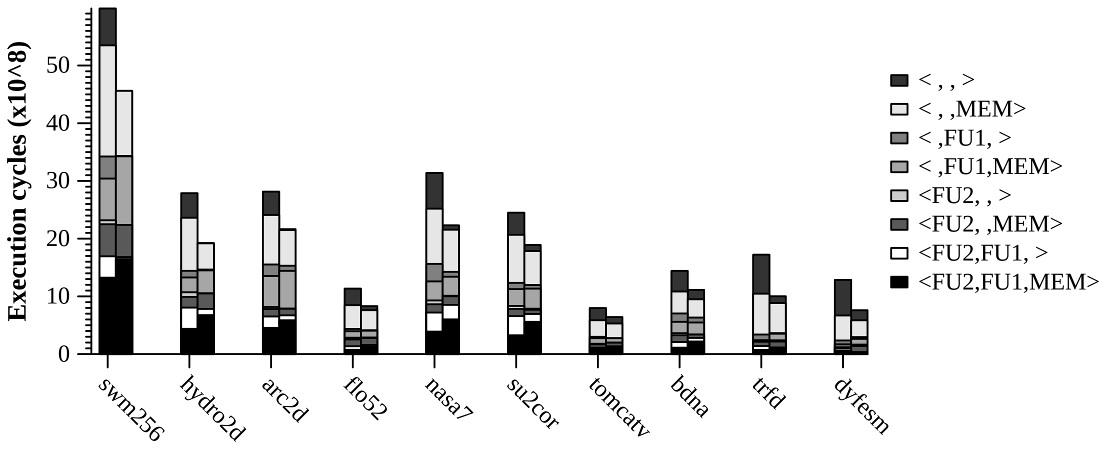

Figure 6.6 Breakdown of the execution cycles for the REF (left bar) and OOOVA (right bar) machines. The OOOVA machine has 16 physical vector registers. For both architectures, memory latency was set at 50 cycles.

of the OOOVA with a fixed memory latency of 50 cycles. In this subsection we consider the ability of the OOOVA machine to tolerate main memory latencies.

Figure 6.7 shows the total execution time for the ten programs when executed on the reference machine and on the OOOVA machine for memory latencies of 1, 50, and 100 cycles. All results are for 16 physical vector registers. As shown in the figure, the reference machine is very sensitive to memory latency. Even though it is a vector machine, memory latency influences execution time considerably. On the other hand, the OOOVA machine is much more tolerant to the increase in memory latency. For most of the benchmarks the performance is flat for the entire range of memory latencies, from 1 to 100 cycles.

Another important point is that even at a memory latency of 1 cycle the OOOVA machine typically obtains speedups over the reference machine in the range of 1.15–1.25 (and goes as high as 1.5 in the case of `dyfesm`). This speedup indicates that the effects of looking ahead in the instruction stream are good even in the absence of long latency memory operations. This might be the case, for example, with a vector multimedia coprocessor or DSP chip operating out of a large L2 data cache.

At the other end of the scale, we see that long memory latencies can be easily tolerated using out-of-order techniques. This indicates that the individual memory modules in the memory

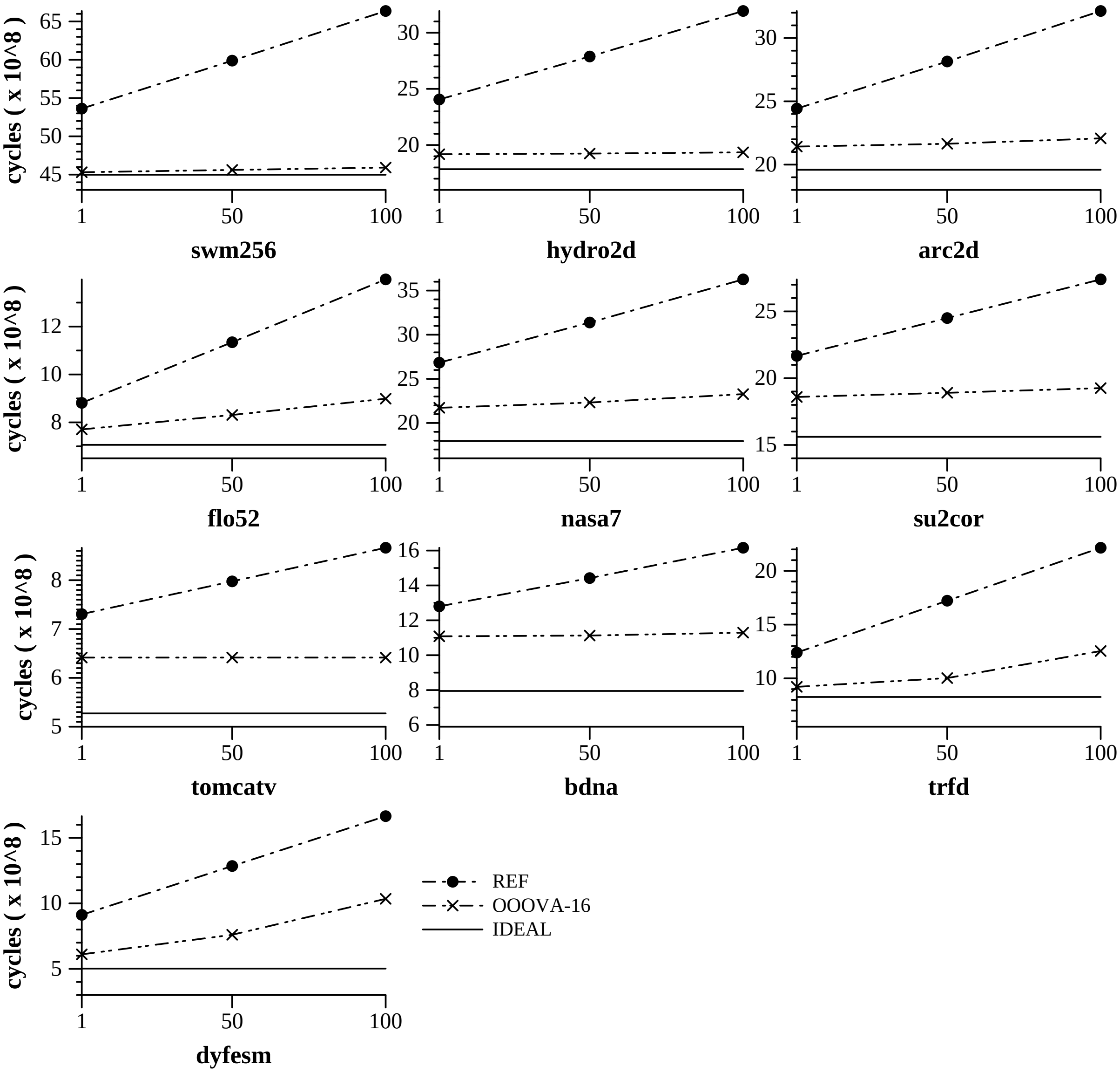

Figure 6.7 Effects of varying main memory latency for the 16 physical vector registers machines.

system can be slowed down (changing very expensive SRAM parts for much cheaper DRAM parts) without significantly degrading total throughput. This type of technology change could have a major impact on the total cost of the machine, which is typically dominated by the cost of the memory subsystem.

6.4 IMPLEMENTING PRECISE EXCEPTIONS

An important side effect of introducing register renaming into a vector architecture is that it enables a straightforward implementation of precise exceptions. In turn, the availability of precise exceptions allows the introduction of virtual memory. Virtual memory has been implemented in vector machines [Con92, UIT94], but is not used in many current high performance parallel vector processors [IW91, Cra90]. Or, it is used in a very restricted form, for example by locking pages containing vector data in memory while a vector program executes [IW91, Tan96].

The primary problem with implementing precise page faults in a high performance vector machine is the high number of overlapped "in-flight" operations – in some machines there may be several hundred. Vector register renaming provides a convenient means for saving the large amount of machine state required for rollback to a precise state following a page fault or other exception. If the contents of old logical vector registers are kept in the reorder buffer until an instruction overwriting the logical register is known to be free of exceptions, then the architected state can be restored if needed.

In order to implement precise exceptions, we introduce two changes to the OOOVA design: first, an instruction is allowed to commit only after it has fully completed (as opposed to the "early" commit scheme we have been using). Second, stores are only allowed to execute and update memory when they are at the head of the reorder buffer; that is, when they are the oldest uncommitted instructions.

The change between early and late commit has important implications on the number of vector registers needed to support out-of-order execution. Early committing enables out-of-order execution with just one extra physical register because it frees an old physical register as soon as a vector instructions start executing. A sequence of three vector instructions can be dispatched on successive cycles because each of them releases one physical vector register that is used by the following instruction. Thus, it is possible to use the three vector functional units in parallel. Late commit, on the other hand, does not permit out-of-order execution with just one extra physical vector register. Instead, it requires one physical register per

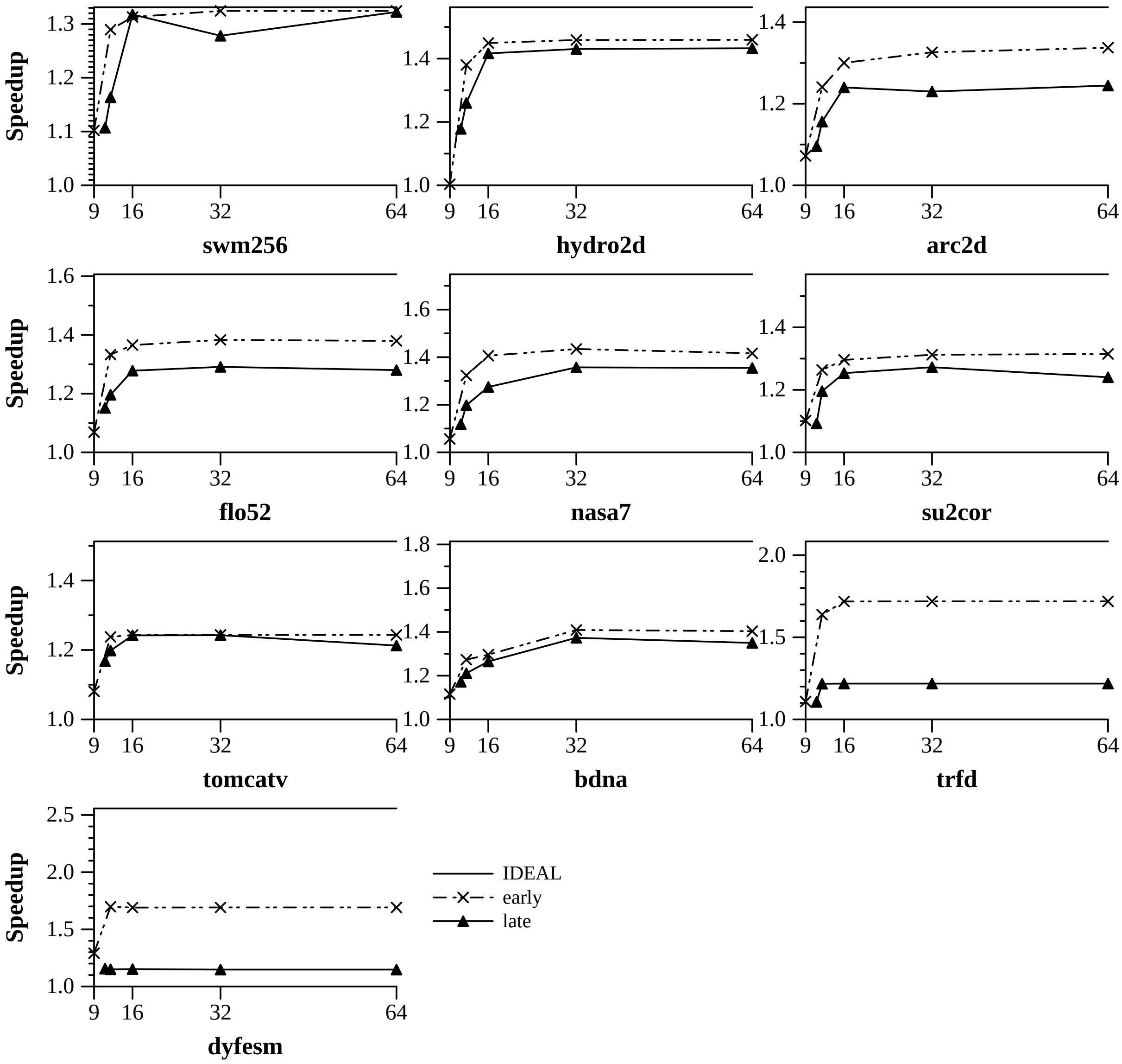

Figure 6.8 Speedups of the OOOVA over the reference architecture for different numbers of vector physical registers under the early and late commit schemes.

architected register plus one extra register per each functional unit that one desires to keep busy. For the OOOVA to achieve an operation per cycle simultaneously for all three vector units, a minimum of $8 + 3 = 11$ physical vector registers are needed.

Figure 6.8 presents a comparison of the speedups over the reference architecture achieved by the OOOVA with early commit (labeled "early"), and by the OOOVA with late commit and execution of stores only at the head of the reorder buffer (labeled "late"). Again, all simulations are performed with a memory latency of 50 cycles.

We can make four important observations about the graphs in Fig. 6.8. First, the performance degradation due to the introduction of the late commit model is small for eight out of the ten programs. Programs `hydro2d`, `arc2d`, `su2cor`, `tomcatv` and `bdna` all degrade less than 5% with 16 physical registers; programs `flo52` and `nasa7` degrade by 7% and 10.3%, respectively. Nevertheless, performance of the other two programs, `trfd` and `dyfesm` is hurt rather severely when going to the late commit model (a 41% and 47% degradation, respectively). This behavior is explained by load-store dependences. As we have already seen in section 5.4.7, the main loop in `trfd` has a memory dependence between the last vector store of iteration i and the first vector load of iteration $i+1$ (both are to the same address). In the early commit model, the store is done as soon as its input data is ready (with chaining between the producer and the store). In the late commit model, the store must wait until 2 intervening instructions between the producer and the store have committed. This delays the dispatching of the following load from the first iteration and explains the high slowdown. A similar situation explains the degradation in `dyfesm`.

Second, in the late commit model, 12 registers are clearly not enough. The performance difference between 12 and 16 registers is much larger than in the early commit model. Thus, from a cost/complexity point of view, the introduction of late commit has a clear impact on the implementation of the vector registers.

Third, even for large numbers of vector registers, the performance curves do not cling to each other. This indicates that late commit has some inherent disadvantage over early commit, apart from the fact that it requires more physical registers. Our analysis show that the reason for the two curves not converging is the policy regarding stores. The fact that store instructions can only update main memory when they become the head of the reorder buffer introduces dependence chains and stalls that are not otherwise present in the early commit scheme.

Fourth, there are some anomalies in the speedups of some programs (`swm256`, `arc2d`, `su2cor` and `tomcatv`) where *increasing* the size of the physical register file actually decreases performance. We investigated these cases and found that the excessive greediness of the out-of-order engine, combined with the instruction ordering in the queues is responsible for the slowdown.

The queues dispatch instructions based not on their age, but on the time the instructions become ready. What happens in `swm256`, for example, with only 16 registers, is that the architecture dynamically unrolls an important basic block 1.5 times. This unrolling is just enough to fill all idle memory cycles and cover up all memory latencies. When given 32 registers, the OOOVA manages to unroll the same loop 2.1 times and is too greedy with its registers. It launches too many loads from the second iteration before launching a store and falls short of registers when it arrives at the loads from the third iteration. At this point, the architecture is forced to waste one full chime of memory activity waiting for a register to become available. Simply changing the dispatch priority to base it on instruction age makes the anomaly disappear (although this has a small negative impact on performance).

6.5 DYNAMIC LOAD ELIMINATION

Register renaming with many physical registers solves instruction issue bottlenecks caused by a limited number of logical registers. However, there is another problem caused by limited logical registers: register spilling. The original compiled code still contains register spills caused by the limited number of architected registers, and to be functionally correct these spills must be executed. Furthermore, besides the obvious store-load spills, limited registers also cause repeated loads from the same memory location. That is, a register may be loaded from memory, but the data can not be saved for long because the register is needed for some other instruction. Then, when the memory data is needed again, it must be re-loaded. In an architecture with more registers, such data can often be held over in a register and does not have to be re-loaded.

Limited registers are common in vector architectures, and the spill problem is aggravated because storing and re-loading a single vector register involves the movement of many words of data to and from memory. Refer to table 3.4, page 45, for a quantitative characterization of spill code in each benchmark.

As we did in chapter 5 for the DVA machine, In this section we propose and study a method that uses register renaming to eliminate much of the memory load traffic due to spills. The method we propose also has significant performance advantages because a load for spilled data is executed in nearly zero time. This is in contrast with the technique used for the decoupled machine, where load elimination did take the time to physically copy data from the output queue back into the input queue.

As opposed to what we did for the DVA, we do not eliminate spill stores, however, because of the need to maintain strict binary compatibility. The memory image should reflect functionally correct state. Relaxing compatibility could lead to removing some spill stores, but we have not yet pursued this approach, since we have already pointed out and defended the advantages of renaming for maintaining a precise state of the processor. We do not want to loose the precise exception property to achieve some performance advantage from store elimination.

6.5.1 Renaming under Dynamic Load Elimination

To eliminate redundant load instructions we propose the following technique. A tag is associated with each physical register (A,S and V). This tag indicates the memory locations currently being held by the register. For vector registers, the tag is a 6-tuple: $\langle @_1, @_2, \mathtt{vl}, \mathtt{vs}, \mathtt{sz}, \mathtt{v}\rangle$, with the same semantic as in chapter 5, page 84, and where v is a validity bit. For scalar registers, the tag is a 4-tuple – vl and vs are not needed. Although the problem of spilling scalar (A and S) registers is somewhat tangential to our study, they are important in the Convex architecture because of its limited number of registers. If the scalar unit had more logical registers, this method would likely not be needed.

Each time a memory operation is performed, its range of addresses is computed (this is done in the second stage of the memory pipeline). If the operation is a load, the tag associated with the destination physical register is filled with the appropriate address information and the valid bit is set. If the operation is a store, then the physical register being stored to memory has its tag updated with the corresponding address information and its valid bit is also set. Thus, each time a memory operation is performed, we "alias" the register contents with the memory addresses used for loading or storing the physical register: the tag indicates an area in memory that matches the register data. As an example, consider a vector double-word load `load 0x800a000-->v2`, performed with VL of 10 and a stride of 16 bytes. The tag "$\langle \mathtt{0x8000a000}, \mathtt{0x8000a098}, \mathtt{10}, \mathtt{16}, \mathtt{8}, \mathtt{1}\rangle$" is associated with the physical vector register to be loaded.

To keep tag contents consistent with memory, when a store instruction is executed its tag has to be compared against all tags already present in the register files. If any conflict is found, that is, if the memory range defined by the store tag overlaps any of the existing tags, these existing tags must be invalidated (to simplify the conflict checking hardware, this invalidation may be done conservatively).

By using the register tags, some vector load operations can be eliminated in the following manner. When a vector load enters the third stage of the memory pipeline, its tag is checked against all tags found in the vector register file. If an exact match is found (an exact match requires all tag fields to be identical), the destination register of the vector load is renamed to the physical register it matches. At this point the load has effectively been completed – in the time it takes to do the rename. Furthermore, matching is not restricted to live registers, it can also occur with a physical register that is on the free list. As long as the validity bit is set, any register (in the free list or in use) is eligible for matching. If a load matches a register in the free list, the register is taken from the free list and added to the register map table.

For scalar registers, eliminating loads is simpler. When a match involving two scalar registers is detected, the register value is copied from one register to the other. The scalar rename table is not affected. Note, however, that scalar store addresses still need to be compared against vector register tags and vector stores need to be compared against scalar tags to ensure full consistency.

A similar memory tagging technique for scalar registers is described in [DC88]. There, tagging is used to store memory variables in registers in the face of potential aliasing problems. That approach, though, is complicated because data is automatically copied from register to register when a tag match is found. Therefore, compiler techniques are required to adapt to this implied data movement. In our application, a tag operation either (a) alters only the rename table or (b) invalidates a tag without changing any register value.

6.5.2 Pipeline modifications

With the scheme just described, when a vector load is eliminated at the disambiguation stage of the memory pipeline, the vector register renaming table is updated. Renaming is considerably complicated if vector registers are renamed in two different pipeline stages (at the decode and disambiguation stages). Therefore, the pipeline structure is modified to rename all vector registers in one and only one stage.

Figure 6.9 shows the modified pipeline. At the decode stage, all scalar registers are renamed but all vector registers are left untouched. Then, all instructions using a vector register pass *in-order* through the 3 stages of the memory pipeline. When they arrive at the disambiguation stage, renaming of vector registers is done. This ensures that all vector instruction see the same renaming table and that modifications introduced by the load elimination scheme

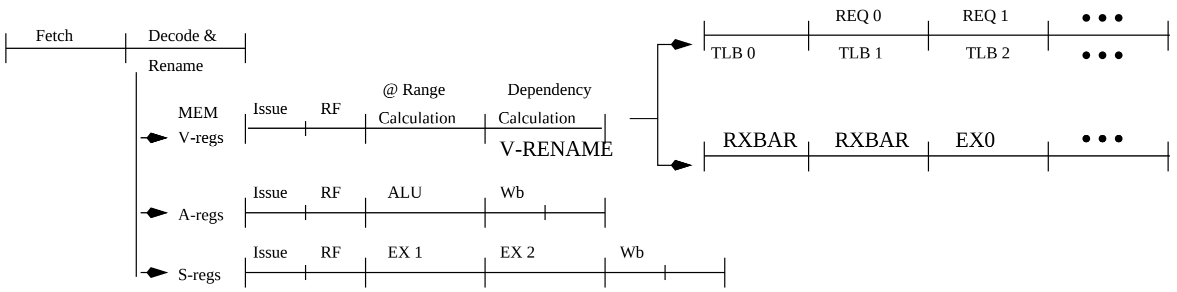

Figure 6.9 The modified instruction pipelines for the Dynamic Load Elimination OOOVA.

are available to all following vector instructions. Moreover, this ensures that store tags are compared against all previous tags in order.

6.5.3 Implementation Notes

There are two details that should be considered when implementing the proposed scheme.

First, allowing two (or more) different entries in the map table that point to the same physical register complicates the register release procedure. In order to avoid prematurely releasing a register, a reference counter is associated with each physical register. When a tag match is found and a new register is mapped to an existing physical register, the counter is incremented. If the physical register is on the free list, the counter is initialized to one. When an instruction in the reorder buffer releases a physical register, its reference count is decremented. If the count drops to zero, the register can be reassigned and is entered into the free list. In the classical renaming algorithm, the reference count never goes beyond 1, and, thus, a true reference count is not needed.

Second, the free list mechanism is modified to maximize the chances for a match. When the register renamer grabs a new register from the free list, it should first try to select a free register that does not have a valid tag. This ensures that the lifetime of a physical register with a valid tag is increased, increasing the probability of a match with a future instruction.

6.5.4 Performance of dynamic load elimination

In this section we present the performance of the OOOVA machine enhanced with dynamic load elimination. As a baseline we use the late commit OOOVA described above, without dynamic load elimination. We also study the OOOVA with load elimination for scalar data only (SLE) and OOOVA with load elimination for both scalars and vectors (SLE+VLE) (in all cases we use the late commit version of the architecture).

Figures 6.10 and 6.11 present the speedup of SLE and SLE+VLE over the baseline OOOVA for different numbers of physical vector registers (16, 32, 64).

For SLE+VLE with 16 vector registers (Fig. 6.11), speedups over the base OOOVA are from 1.04 to 1.16 for most programs and are as high as 1.63 and 1.71 for `dyfesm` and `trfd`. At 32 vector registers registers, the available storage space for keeping vector data doubles and allows more tag matchings. The speedups increase significantly and their range for most programs is between 1.10 and 1.20. For `dyfesm` and `trfd`, the speedups remain very high, but do not appreciably improve when going from 16 to 32 registers.

Doubling the number of vector registers again, to 64, does not yield much additional speedup. For most programs, the improvement is below 5%, and only `tomcatv` and `trfd` seem to be able to take advantage of the extra registers (`tomcatv` goes from 1.16 up to 1.30). The results show that most of the data movement to be eliminated is captured with 32 physical vector registers.

The remarkably different performance behavior of `dyfesm` and `trfd` requires explanation. This can be done by looking at SLE (Fig. 6.10). Under SLE, all other programs have very low speedups (less than 1.05) and, yet, `trfd` and `dyfesm` achieve speedups of 1.30 and 1.36, respectively (for the configuration with 32 vector registers). Our analysis of these two programs shows that the ability to bypass scalar data allows these programs to "see" more iterations of a certain loop at once. Consider a vector loop where the *scalar* register pressure is high. At the top of loop there is a scalar load for variable K into register a5. This load is on the critical path because register a5 will be used as the base address for following vector memory operations. Toward the end of the loop, the compiler updates a5 and stores it back into variable K (this loading and later storing, while not qualifying as pure spill code is a direct consequence of the high register pressure mentioned). Without SLE, the load at the start of the loop will conflict with the store of the previous iteration. The load will wait for the store to reach the head of the reorder buffer before executing. This will only happen after

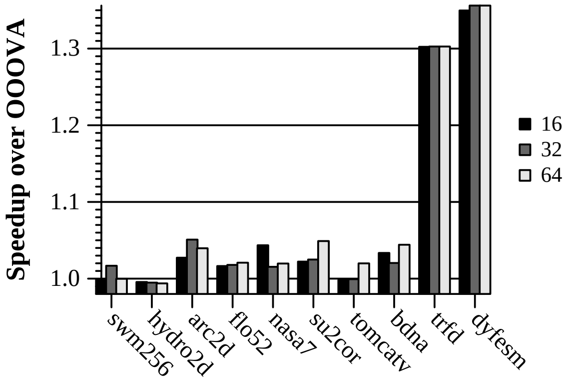

Figure 6.10 Speedup of SLE over the OOOVA machine for 3 different physical vector register file sizes.

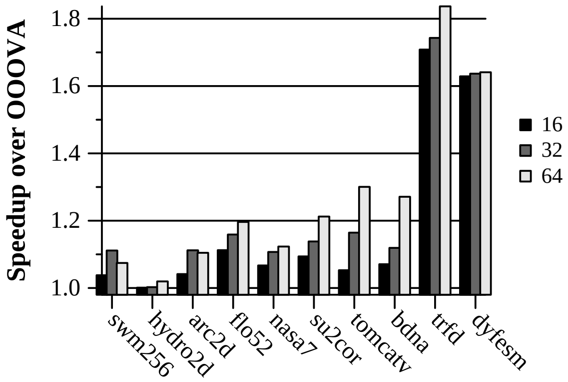

Figure 6.11 Speedup of SLE+VLE over the OOOVA machine for 3 different physical vector register file sizes.

all previous vector stores have successfully retired, and thus, the previous iteration will be almost complete by the time the store executes. Since the scalar load is needed by all following vector memory instructions, no memory operations in the new loop iteration will be dispatched until the previous iteration is (almost) complete. This reduces the amount of ILP exposed to the architecture and hurts performance. When SLE is added, the scalar load can bypass its data right away, and two (or more) iterations of the loop can be overlapped. This overlapping covers most idle cycles of the memory port and produces the high speedups.

6.5.5 Caching Loads to improve DLE

So far we have been exploiting the fact that loads can match the tag of previous stores. Nonetheless, there is nothing in the implementation that prevents loads to be also tagged. That is, when a load is sent to memory, its "memory region" is computed and stored in the corresponding tag. If, later, another load instruction to the same exact memory region is executed we could have a matching between two tags belonging to two loads. We refer to this load-to-load matching as "caching loads" (CL for short).

Allowing this load-to-load matching can be of benefit if there are repeated loads from the same memory position. Going back to table 3.4 and looking at column "Avg. Reuse" we see that in most programs a spilled variable is only reloaded once, except in `arc2d`, `su2cor` and `bdna`, where the reuse is higher. Nonetheless, there is a form of spill code that we have not been able to measure in which the compiler repeatedly re-loads a variable because of not having enough registers to keep it inside the processor. This load-to-load matching will catch this kind of hidden spill as well as a small fraction of the temporal locality that programs might have.

Nonetheless, allowing load-to-load matching represents a tradeoff between two opposed effects:

- Without CL, there is less pressure on the number of registers available for holding store data. All physical registers, even those on the free list, can potentially be devoted to be "aliased" with store data. This should increase the probability of load-to-store matchings, because it indirectly enlarges the lifetime of registers that have been marked valid through stores.
- When CL is enabled, the physical registers can now either hold load data or store data. Load-to-load matchings are now possible, but we are (potentially) diminishing the amount of registers that hold store data. When we use a register to hold load data, we might be kicking out store data that typically has more chances of being matched.

Figure 6.12 presents the extra speedup achieved over the SLE+VLE configuration when we allow load-to-load matchings. Again, we present the speedups for three different register file sizes. Overall, we see that the CL feature provides significant improvements over the SLE+VLE configuration. This is counterintuitive since (a) the initial goal of virtual spilling was to reduce spill loads and (b) it is generally accepted that vector applications have data sets too large to fit even in large caches, let alone in only 32 or 64 registers. As already mentioned,

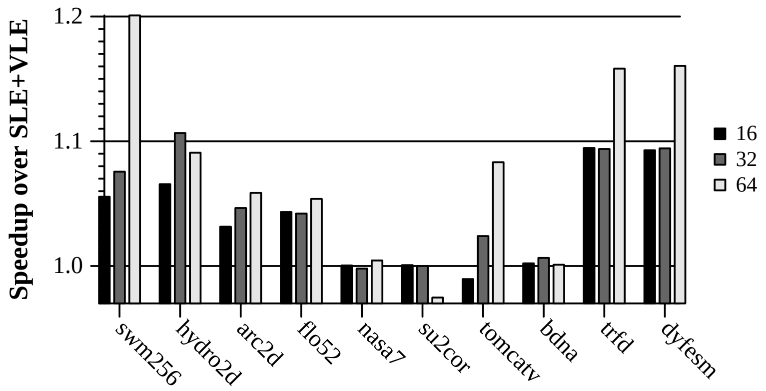

Figure 6.12 Extra speedup of SLE+VLE+CL over SLE+VLE when adding load-to-load matchings. Each bar represents a different vector register file size.

the explanation is a combination of what we term "hidden spill" and some caching effect. With only 8 vector registers, the compiler is typical forced to load and reload the same variable several times in a loop. This, without being commonly termed as spill (there is no store and later re-load), it is also a direct consequence of the high register pressure. Our mechanism allows that a second load to the same variable is discarded and performed by a simple rename table change. Undoubtedly, some programs are also seeing a caching effect of variables loaded in a certain iteration of a loop and reloaded later on. We have not yet been able to quantify the relative importance of these two effects, although the hidden spill seems the more likely cause of success of the SLE+VLE+CL configuration.

We finally note that, if dynamic load elimination is implemented in a processor, then the load-to-load matching would be always implemented. That is, the extra cost of CL over the previous configurations is negligible and yet produces a substantial amount of speedup.

6.5.6 Traffic Reduction

A very important effect of dynamic load elimination is that it reduces the total amount of traffic seen by the memory system. On top of the performance improvements shown by dynamic load elimination, the traffic reduction would be very beneficial in multiprocessing environments. The less load on the memory system, the more likely it is that bottlenecks in the interconnection network and in the memory modules will be avoided. From these two effects, an overall system performance improvement can be expected.

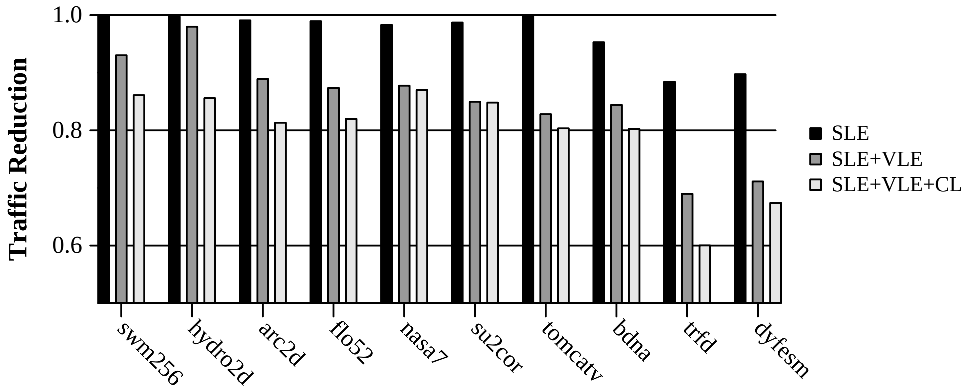

Figure 6.13 Traffic reduction under dynamic load elimination with 32 physical vector registers.

We have computed the traffic reduction of each of the programs for the two dynamic load elimination configurations considered. We define the traffic reduction as the ratio between the total number of requests (load and stores) sent over the address bus by the SLE, SLE+VLE or SLE+VLE+CL configurations divided by the total number of requests done by the baseline OOOVA. Figure 6.13 present this ratio for 32 physical vector registers. As an example, Fig. 6.13 shows us that the SLE configuration for `dyfesm` performs 11% fewer memory requests than the OOOVA configuration.

As can be seen, for SLE+VLE+CL, the typical traffic reduction is between 15 and 20%. Programs `dyfesm` and `trfd`, due to their special behavior already mentioned, have much larger reductions, as much as 40%.

6.6 RELATED WORK

There is little related work regarding out-of-order execution in the context of vector processors. To the best of our knowledge, this is the first published work on the subject.

Most work on out-of-order execution and register renaming has concentrated in superscalar processors. There is abundant literature describing current superscalar processors that have a fully out-of-order engine with register renaming. Diep et al. [DNS95] describe the architecture of the PowerPC 620, a 4-wide superscalar machine with six execution units and register renaming to support out-of-order execution. Barreh et al. [BDHS94] describe the Power2 with similar features. The MIPS R1000, a 4-way superscalar using out-of-order execution and register renaming, is described in [Yag96]. Similar designs are the HP PA-8000 [Hun95], the SPARC64 MCM [WPS95]. The recent ALPHA 21264 [Kel96] has also out-of-order execution and register renaming, mapping 32 logical onto 80 physical floating point registers and mapping 32 logical onto 72 physical integer registers.

We also note that most work in superscalar processor architecture design is not directly relevant to the work being presented, since the OOOVA, although out-of-order, is a single issue machine. Thus, it does not try to execute more than one instruction per cycle in the superscalar sense. Rather, multiple operations per cycle are achieved because long vector instructions are overlapped one with each other.

6.7 SUMMARY

In this chapter we have considered the usefulness of out-of-order execution and register renaming for vector architectures. We have seen through simulation that the traditional in-order vector execution model is not enough to fully use the bandwidth of a single memory port and to cover up for main memory latency (even considering that the programs were memory bound). We have shown that when out-of-order issue and register renaming are introduced, vector performance is increased. This performance advantage can be realized even when adding only a few extra physical registers to be used for renaming. Out-of-order execution is as useful in a vector processor as it is widely recognized to be in current superscalar microprocessors.

Using only 12 physical vector registers and an aggressive commit model, we have shown significant speedups over the reference machine. At a modest cost of 16 vector registers, the range of speedups was 1.24–1.72. Increasing the number of vector registers up to 64 does not lead to significant extra improvements, however.

Moreover, we have shown that large memory latencies of up to 100 cycles can be easily tolerated. The dynamic reordering of vector instructions and the disambiguation mechanisms introduced allow the memory unit to send a continuous flow of requests to the memory system. This flow is overlapped with the arrival of data and covers up main memory latency.

The introduction of register renaming gives a powerful tool for implementing precise exceptions. By changing the aggressive commit model into a conservative model where an instruction only commits when it (and all its predecessors) are known to be free of exceptions, we can recover all the architectural state at any point in time. This allows the easy introduction of virtual memory. Our simulations have shown that the implementation of precise exceptions costs around 10% in application performance, though some programs may be much more sensitive than others.

One problem not solved by register renaming is register spilling. The addition of extra physical registers, per se, does not reduce the amount of spilled data. We have introduced a new technique, dynamic load elimination, that uses the renaming mechanism to reduce the amount of load spill traffic. By tagging all our registers with memory information we can detect when a certain load is redundant and its required data is already in some other physical register. Under such conditions, the load can be performed through a simple rename table change. Our simulations have shown that this technique can further improve performance typically by factors of 1.07–1.16 (and as high as 1.78). The dynamic load elimination technique can benefit from more physical registers, since it can cache more data inside the vector register file. Simulations with 32 physical vector registers show that load elimination yields improvements typically in the range 1.10–1.20. Moreover, at 32 registers, load elimination can reduce the total traffic to the memory system by factors ranging between 15–20% and, in some cases, up to 40%.

Finally, we feel that our results should be of use to the growing community of processor architectures implementing some kind of multimedia extensions. As graphics coprocessors and DSP functions are incorporated into general purpose microprocessors, the advantages of vector instruction sets will become more evident. In order to sustain high throughput, both from and to special purpose devices such as frame buffers, long memory latencies will have to be tolerated. These types of applications generally require high bandwidths between the chip and the memory system not available in current microprocessors. For both bandwidth and latency problems, out-of-order vector implementations can help achieve improved performance.

7

MULTITHREADED VECTOR ARCHITECTURE

Summary

In this chapter we exploit an orthogonal solution to the memory latency and bandwidth problems. Instead of speeding up programs individually, we extract instruction level parallelism by running together several vectorized codes, in a multithreaded way. Multithreading techniques can be applied to a vector processor as easily as in scalar processors to greatly increase processor throughput and maximize resource utilization. Among the techniques seen so far, multithreading provides the best occupation ratios on the memory bus (95%), although causes some individual program slowdowns. This chapter also looks at the costs associated with the replication of the vector register file required for multithreading.

7.1 INTRODUCTION

Multithreaded scalar processors [Aga92, TEE+95, TEL96, HKN+92, EJK+96] attack the memory latency problem by switching between threads of computations. This switching can be made every time a long-latency operation (such as a cache miss) threatens to halt the processor or, alternatively, can be done on a cycle-by-cycle basis. Multithreading not only fights memory latency, but also produces a system with higher throughput and better resource utilization. The change of thread on long latency operations implies an increase in exploitable parallelism, a decrease on the the probability of halting the cpu due to a hazard, and yields a higher occupation of the functional units and an improved system throughput. While each single thread still pays latency delays, the cpu is (presumably) never idle thanks to this mixing of different threads of computation.

Multithreading can be applied to achieve high performance in two orthogonal directions. By using a specialized compiler and, maybe, some special instructions, it is possible to extract independent threads from a program and run them concurrently. Hopefully, these threads will run in parallel on the underlying multithreaded machine and cooperate towards the completion of the original program. By switching from thread to thread on long latency operations, the individual latencies are masked, the resource usage is maximized and the program completes in less time than on a sequential machine. On the other hand, multithreading can be used to improve the global *throughput* of a machine; that is, instead of focusing on speeding up a single program, the improvement is on number of jobs completed per unit time. In a sense, both styles of multithreading provide increased performance, but they focus on slightly different workloads.

The purpose of this chapter is to show that using multithreading techniques in a vector processor, the efficiency of a vector processor can be improved greatly. In particular we will show how occupations of up to a 95% of a single memory bus can be obtained. We will also show how, even for an ideal memory system with no latency, multithreading provides a significant advantage over standard mode of operation. We will also present data showing that for more realistic latencies, multithreaded vector architectures perform substantially better than non-multithreaded vector architectures. This work has focused on the throughput style of performance, and we will take as our base performance measure the total execution time of a set of several programs.

The cost of multithreading a vector processor is in general terms, relatively high, since it requires to replicate an expensive resource: the vector register file and the read/write register crossbars. Roughly speaking, this cost would be higher than the queues added to the DVA machine and would be in the same order of magnitude as the extra physical registers added to the OOOVA machine. In contrast, multithreading a superscalar processor would be relatively more simple, since the total amount of area to be replicated would be much smaller.

While replicating the register file is expensive, multithreading keeps control complexity very low. The only extra hardware needed is a set of N program counters and some logic to be able to switch from thread to thread. Compare this with the renaming hardware and the pipeline stage introduced in the OOOVA only for the rename phase, or compare it with the instruction splitter found in the DVA, where we needed to replicate the instruction decoder for each independent processor.

Finally we would like to point out that multithreading is orthogonal to out-of-order execution and decoupling. That is, in this chapter we will apply multithreading directly to the reference machine, but straightforward extensions would be to apply it to the OOOVA machine or to the DVA machine. Combining multithreading and out-of-order execution yields a system where several instructions from different threads can be active eat the same time inside the processor [YN95, TEE$^+$95]. In such a configuration the out-of-order engine described in chapter 6 would be kept unmodified and a multithreaded fetch unit would be added to it. Renaming provides a very convenient way to implicitly distinguish instructions from different threads and the out-of-order engine automatically maximizes resource usage among a set of different threads.

Combining decoupling and multithreading would also be straightforward. By replicating the input/output queues or, alternatively, by sharing them among different threads one could get the best of both worlds: latency tolerance in individual threads thanks to decoupling and resort to a thread switch when a thread is about to block, thereby taking advantage of the latency tolerance properties of multithreading.

7.2 THE MULTITHREADED VECTOR ARCHITECTURE (MVA)

The multithreaded version of the reference architecture (MVA) is shown in Fig. 7.1. It has several copies of all three set of registers (A, S and V) needed to support multiple hardware contexts (up to a maximum of 4 contexts). The fetch and decode units are essentially the same as in the reference machine, except that they are time multiplexed between the N contexts in the machine. At each cycle, the decode unit looks at one and only one thread. If the current instruction of that thread can be dispatched, the instruction is sent to its functional unit and the fetch unit is signaled to start fetching the following instruction from that thread. If the instruction can not proceed, then this decoding cycle has been lost and the switch logic will select some other thread to attempt decoding in the following cycle. Therefore, this scheme can dispatch at most 1 instruction per cycle.

There are no special out-of-order or simultaneous issue features in our multithreaded architecture. At most one instruction is fetched per cycle and at most one instruction is sent to the execution units per cycle. Moreover, instructions from within a single thread execute in-order, exactly the same way as they would do on the reference processor. The lack of sophisticated issue units greatly simplifies the overall design.

The key point in the design of the multithreaded vector architecture is the vector register file. The other two register files (A and S) only hold 8 registers each and thus only 32 scalar registers per file would be needed to support 4 contexts. Much larger register files are commonplace and thus we will not consider this part of the architecture as an important concern. On the other hand, the vector register file holds a total of $8 \times 128 = 1024$ 64-bit registers and introducing more contexts increases significantly the area needed to implement the processor. Not only that, the cost of the read/write crossbars between the registers and the functional units is another limiting factor that has to be taken into account. In our model we assume that we completely replicate the read/write crossbars, so that each thread sees the same amount of connectivity as it had in the reference architecture. For 4 contexts, this assumption implies a read crossbar of 32 by 3 and a write crossbar of 3 by 16. Similarly, at 4 contexts we have a total of 4096 64-bit registers (32Kb), which is in the range of current primary data caches. While these extra costs are not unsurmountable, we will take them into account in our study and charge the multithreaded processor with an extra cycle both for reading and writing from/into the vector register file.

The baseline scheduling among threads we will use is as follows: We allow each thread to run (fetch and issue) until it blocks on some data dependency or some resource conflict. When a thread blocks, we chose some other thread that is known not to be blocked. If several of those exist, we always choose the lowest-numbered thread (unfair scheme). The reason we allow a thread to proceed as long as it does not block (as opposed to switching threads every cycle, for example) is to favor the amount of chaining happening between vector instructions. The unfair scheme where we define thread 0 as being the highest priority thread is chosen to

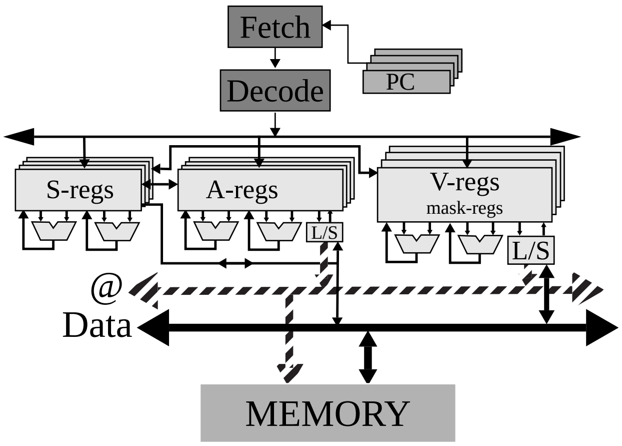

Figure 7.1 The Multithreaded vector architecture.

try to have at least one thread that will not experience a severe slowdown when run together with other vector threads.

Machine Parameters

All machines parameters referring to the MVA are the same as already presented in section 2.4. The only significant difference is that we need to take into account the high number of vector registers present in some of the configurations. This high number of registers can put in jeopardy the cycle time due to the read/write crossbar loading and might require adding one cycle of extra latency to read and write operands.

7.3 SIMULATION STRATEGY

Our benchmark programs (discussed in 2.6) are run through the multithreaded simulators in groups of two, three or four programs depending on the number of hardware contexts that the architecture being simulated has. Since the 10 programs we have selected are of different run-length, when run together on the multithreaded machine they may complete at very different times. In order to ensure that each benchmark is run to completion at least once, we have taken the following approach (see Fig. 7.2): when a group of programs is run, the simulator

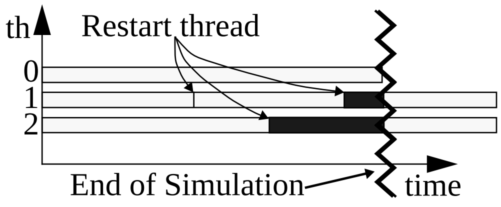

Figure 7.2 An example run of a multithreaded simulation with 3 hardware contexts.

will run until the program that has been allocated to hardware context 0 completes. If the programs that run on some other context are much shorter and complete before thread 0 finishes, then we restart them as many times as necessary. When program 0 completes, the other programs running are in some intermediate point in their computations. For example, in Fig. 7.2, program 0 runs to completion. In that amount of time, program 1 has run twice to completion and has been able to execute a fraction (f_1) of its third run. Similarly, program 2 has run once to completion and has executed a fraction (f_2) of its second run.

To compute speedups, we need the time taken by the reference architecture to execute *the same amount* of work as the multithreaded version. We run through the reference simulator all three programs (each of which takes C_i cycles to complete), we run program 1 up to point f_1 (which takes F_1 cycles to complete), and we run program 2 up to point f_2 (which takes F_2 cycles to complete). If the multithreaded run has taken T cycles to complete, we compute speedup as

$$(C_0 + 2 \times C_1 + C_2 + F_1 + F_2)/T \tag{7.1}$$

This speedup measure will be used throughout the rest of this chapter.

The grouping of programs was done as follows. Since running all combinations of 10 programs in groups of 2, 3 and 4 was too costly in terms of simulation time, we selected a subset of all permutations in a pseudo-random manner. We selected (randomly) 5 programs for the 2 thread experiments (shown in first column of table 7.1), 2 programs for the 3 threads experiments (column 2) and 1 last program for the 4 thread experiments (column 3). Now, in order to compute the speedup associated with program X one would run: 5 simulations grouping program X with each program appearing in column labeled "2" of table 7.1 (example: X+`flo52`); 10 simulations grouping program X with all 5 programs from column 2 and the 2 programs in column 3 (example: X+`bdna`+`swm256`); and, finally, 10 simulations grouping

Prog.	Num. Threads		
	2	3	4
X	+bdna +su2cor +tomcatv +swm256 +flo52	+swm256 +hydro2d	+arc2d

Table 7.1 Randomly selected programs to form groupings.

program X with all 5 programs from column 2, the 2 programs from column 3 and the program in column 4 (example: X+`tomcatv`+`hydro2d`+`arc2d`).

These scheme, while not complete, gives us enough data points to detect outliers and cases where two programs complement each other so well that produce higher than usual speedups.

7.4 PERFORMANCE OF THE MULTITHREADED VECTOR ARCHITECTURE

In this section we present the performance of the multithreaded vector architecture versus the reference architecture. We use three related metrics to evaluate the merits of multithreading: speedup, memory port occupation and vector operations performed per cycle (VOPC). Speedup has already been defined in section 7.3. Memory port occupation is defined as the total number of memory requests sent over the address bus divided by total number of cycles. Since we only have one address port, this metric ranges from 0 to 1. Vector operations per cycle is defined as total number of vector operations performed divided by total number of cycles. Our machine has two vector arithmetic units so this metric will vary between 0 and 2.

For each program and for each metric we plot the average of all simulation runs of a given program. For example, the data point corresponding to program `hydro2d` and 2 hardware contexts in Fig. 7.4 is the average memory port utilization computed across all five `hydro2d` simulations (`hydro2d` plus itself, `bdna`, `su2cor`, `tomcatv` and `swm256`).

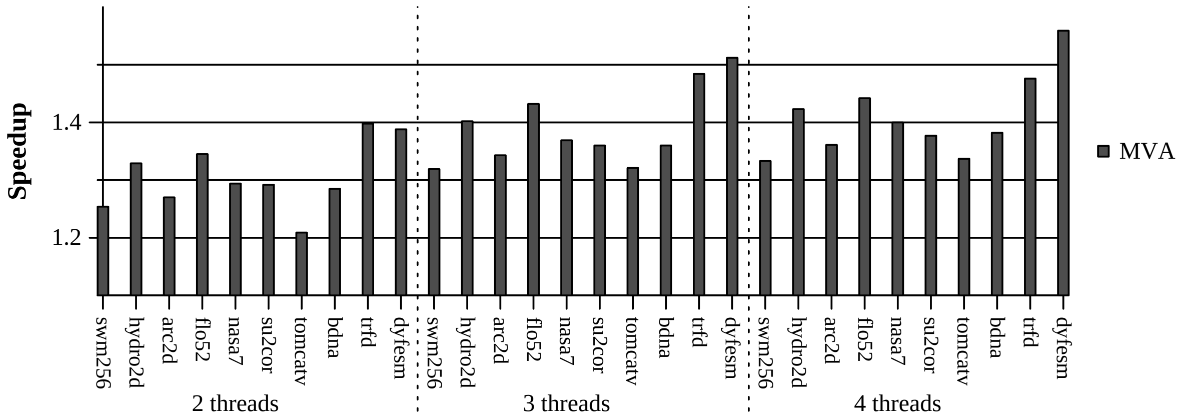

Figure 7.3 Performance of the Multithreaded approach for 2, 3 and 4 contexts.

7.4.1 Speedup

The effects of multithreading the reference architecture can be seen in Fig. 7.3. The memory latency for the experiments shown in these figures was set at 50 cycles. Each figure shows the speedup obtained by each of the ten benchmark programs when grouped with some other programs from the suite, as described in section 7.3.

From this figure it can be seen that typical speedups with just two vector contexts range between 1.2 and 1.4. Going to 3 vector contexts, the multithreaded architecture consistently sustains a speedup of 1.3 and it can go as high as 1.51. The increase in performance for going from three contexts up to four contexts is much lower but still significant.

It is interesting to note that the higher speedups correspond to programs (`dyfesm` and `trfd`) that make both a low utilization of the memory bus and the functional units. When these programs are run on the multithreaded machine, they leave a lot of “empty holes” where it is easy that other programs can fit. Thus, the total speedup achieved is larger because all the wasted cycles of these two programs are profitably reused running other threads.

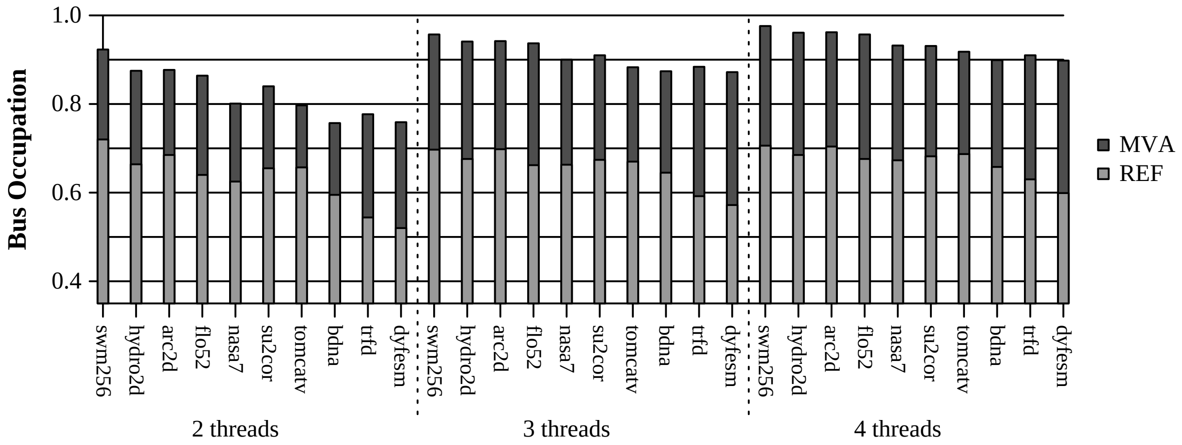

Figure 7.4 Occupation of the memory port for 2, 3 and 4 contexts for the MVA and REF architectures.

7.4.2 Memory Port Usage

The utilization of the single memory port can be seen in Fig. 7.4. What we plot in this figure (bars labeled "MVA") is the number of cycles where the address port was busy over the total execution cycles, on the multithreaded machine. Numbers close to 1 indicate that the memory port is close to saturation and that, in a sense, and given the programs as they are, we are close to the optimal speed we can achieve. In order to compare with the usage of the memory port in the reference architecture, the figure also presents the average utilization of the memory bus achieved by each tuple of programs when run sequentially on the reference machine (bars labeled "REF").

Figure 7.4 shows how for the 2 contexts architecture, typical usage of the memory port is around 80%. This value is in sharp contrast to the utilization in the non-multithreaded architecture. For example, if we run program `flo52` and its 5 companions sequentially on the reference machine we would see an average utilization of the memory bus of around 60%, yet when these programs are run in pairs on the multithreaded machine, the average utilization goes up to an 86%.

Similarly, having three contexts achieves a 90% utilization of the memory port for almost all programs. Note that reaching higher levels of utilization is hard since there are several factors that lower the utilization that multithreading can not cover up. In the first place, our programs have a scalar portion of their code that ranges between 0.1% up to 25%. When

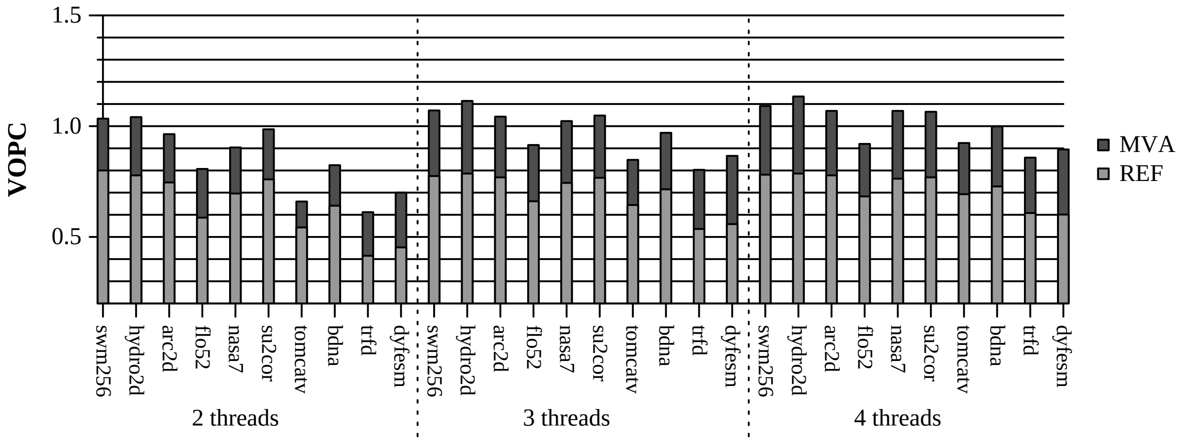

Figure 7.5 Occupation of the vector functional units for 2, 3 and 4 contexts for the MVA and REF architectures.

one thread is inside a scalar loop it will seldom reach a memory utilization higher than 1/3. Typically one of this loops has a load and a store plus at least two address calculation instructions, a compare, a branch and maybe some other computation instruction (recall that the reference machine only has eight scalar registers and thus the compiler does not apply aggressive techniques such as unrolling or software pipelining). This yields 2 memory operations per 6–8 instructions. Since these instructions are executed one per cycle, the memory occupation is somewhat bounded. This factor can easily be seen in Fig. 7.4 by noting that the bus occupation goes down when we move to the right end of the 2 contexts bars, the 3 contexts bars and the 4 contexts bars, where the less-vectorized programs lie. In the second place, latencies and conflicts inside the vector processor also degrade the rate at which memory instructions are dispatched to the memory system. Latencies are typically hidden through multithreading but resource unavailability is harder to cover up.

7.4.3 Vector Operations per cycle

Our third metric of performance for the multithreaded vector architecture is the number of vector arithmetic operations performed per cycle. This measure is defined as the sum of all operations performed on the two vector computational units divided by the total execution time. This measure is close to flops per cycle (a more common metric found in the literature) but since there is a significant fraction of computations done in vector mode that are fixed point, the metric chosen better reflects the total amount of work done by the vector unit.

Figure 7.5 presents the VOPC metric for the three multithreaded architectures under study and for the reference architecture. The first point to note is that for the reference machine, VOPC is in the range between 0.5 and 0.85. These low values are not surprising given what we have seen in section 7.4.2. If the reference machine only performs between 0.5 and 0.8 memory transactions per cycle, it is expected that it will be difficult to sustain even one flop per cycle. To do so would require a larger degree of reuse inside the processor which, in turn, would require a larger number of registers.

For two contexts, the top 6 most-vectorizable programs reach around 1 vector operation per cycle, while the remaining four (`tomcatv`, `bdna`, `trfd` and `dyfesm`) increase a lot their VOPC but still stay below 0.9 in almost all cases. For three contexts, VOPC typically exceeds 1.0 for the six most vectorizable programs. Again, the remaining four can be split in two distinct groups: `tomcatv` and `bdna`, when joined with one of the highly vectorizable codes, barely manage to reach 1.0, while `trfd` and `dyfesm` make a very light use of vector functional units. Improving this numbers is difficult, as Fig. 7.5 shows, because although we could increase to four hardware contexts, the memory bus is already almost saturated (around 90% usage) and determines the maximum throughput of the system.

7.5 VARYING MEMORY LATENCY

One of the abilities of multithreading is to hide long memory latencies. In previous sections we have seen the benefits of multithreading from the point of increased throughput. This section will look at how well does multithreading tolerate increasing main memory latency. All results presented so far have assumed a fixed memory latency of 50 cycles. In this section we will present the effects of varying this memory latency between 1 and 100 cycles.

In order to present the data, we will need to change our benchmarking strategy. To evaluate the effects of memory latency on our multithreaded processor, we need to fix the total amount of work being simulated. Recall that the simulations being shown in previous section were finished when thread 0 did complete. If we execute two simulations of programs `swm256` (on thread 0) and `tomcatv` (on thread 1) on a 2 context processor setting memory latency at 50 and 100 cycles, for example, in each simulation the amount of work completed for thread 0 will be the same, but not the work completed for thread 1. This makes difficult the comparison of speedups or memory usage per cycle since the two runs have performed slightly different tasks.

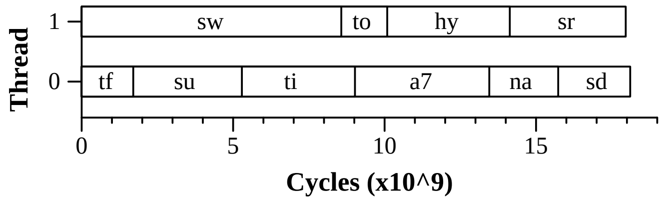

Figure 7.6 Execution example of the 10 programs run on a 2-context machine (memory latency 50 cycles).

To overcome this problem, we will define a new benchmark consisting in the execution of *all* 10 programs used so far. These 10 programs are ordered randomly (in particular, the order chosen is: `flo52`, `swm256`, `su2cor`, `trfd`, `tomcatv`, `nasa7`, `hydro2d`, `bdna`, `arc2d`, `dyfesm`). Then, when doing simulations of a, say, 3 context processor each thread is initialized to a different program from this list sequentially. When a thread completes, the next job from the list is assigned to that thread. Using this scheme, which is the same used in [HKN+92], the amount of work performed is always fixed. The only shortcoming of this technique is that towards the end of a simulation run, some hardware context might end up being empty, and thus not all the potential performance improvement can be realized. Figure 7.6 shows an example execution profile of the 10 programs on a 2-context architecture. Note how towards the end, program `dyfesm` is for a short period of time alone on the machine.

Figure 7.7 presents the total execution time of our ten programs when executed on the reference machine and on the multithreaded machine with 2, 3 and 4 contexts. For the reference data, each single data point represents the execution time of the 10 programs when executed sequentially one after the other. On the multithreaded machines, each data point corresponds to the execution of the ten benchmarks using 2, 3 or 4 hardware contexts. We have also included a line (labeled IDEAL) that indicates the lowest possible execution time, computed by removing all data dependencies from the programs and looking only at the most saturated resource and taking the utilization of that resource as the lower bound for execution time.

As it can be seen from this figure, the reference machine is very sensitive to memory latency. Even being a vector machine, memory latency influences execution time almost linearly. On the other hand, the multithreaded machine is much more tolerant to the increase in memory latency. The curve for 2 contexts, for example, is relatively flat. At latency 1, the speedup of the multithreaded architecture over the reference machine is 1.15 and at latency 100 the speedup goes up to 1.45. Moreover the relative degradation in performance of the 2 contexts

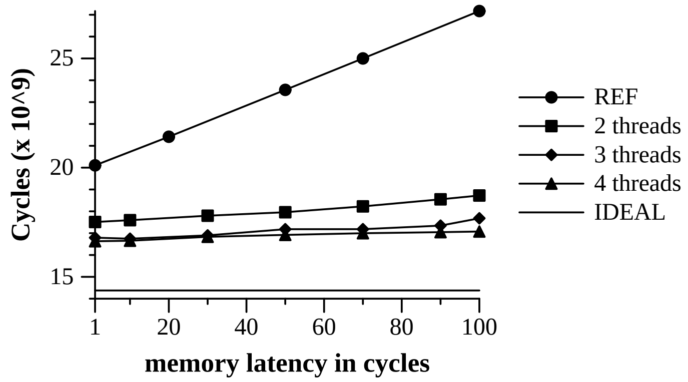

Figure 7.7 Comparison of total execution time of the 10 benchmarks when run on the reference machine and on the multithreaded architecture with 2, 3 and 4 contexts, when varying memory latency.

architecture is low, with the difference between the execution time at a latency of 1 cycle and a latency of a 100 cycles being only a 6.8%.

A second important point is that even at a memory latency of 1 cycle the multithreaded machine obtains a speedup of 1.15 over the reference machine. Such a low value of memory latency represents an ideal memory system which is very far from current values of memory latency (a Cray C90 machine has around 23 cycles of main memory latency). This speedup indicates that the effects of interleaving different threads are good even in the absence of long latency memory operations. The interleaving increases the usage of critical resources and allows two or more threads to progress in their computations together.

An interesting effect of the latency tolerance properties of multithreading in the context of vector processors is that the memory system could be slowed down without significantly degrading total throughput. This slow down could be achieved by using slower DRAM parts instead of the currently very expensive SRAM chips found in typical vector supercomputers. This type of technology change could have a major impact on the total cost of the machine, which is typically dominated by the cost of the memory subsystem. The reduction in cost would largely compensate the extra cost in the cpu component introduced by the different copies of the vector registers.

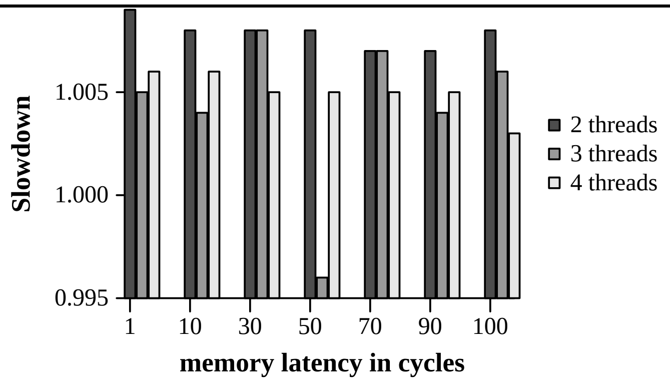

Figure 7.8 Slowdown due to increasing the latency of the vector registers read/write crossbars from 2 to 3 cycles.

7.6 EFFECTS OF INCREASING THE SIZE OF THE REGISTER CROSSBARS

Data presented so far has used a 2 cycle latency for the vector register file read/write crossbars. As noted in section 7.2, replicating the vector register file has a significant impact on cost, area and speed. We are mostly concerned by the speed implications. As already mentioned, going to 4 contexts implies a much larger crossbar both for reading and for writing the vector register file. It is reasonable to accept that this larger crossbar would take an extra cycle to be crossed than the crossbar found on the reference machine.

In this section we study the effect of adding an extra cycle to the read/write crossbars (for a total of 3 cycles in both cases) on total execution time. Figure 7.8 presents the slowdown experienced by our combined benchmark of 10 programs for different values of latency. Overall, the slowdown is lower than 1.009. This is not surprising since three factors mitigate the effect of the extra cycle added: first, a vector machine by itself has a granularity large enough to tolerate small increases in latency. Second, the machine is multithreaded, which also helps in covering up the extra cycle. Finally, chaining also contributes to making the slowdown minimal.

There is a surprising case at latency 50 and 3 threads, where the benchmark actually runs faster. This is due to the fact that the increase in latency makes program `nasa7` take slightly longer to complete. In turn, this induces that the two following programs in the suite, `bdna` and `arc2d` interchange their positions from thread 0 to thread 1 and vice-versa. Since our

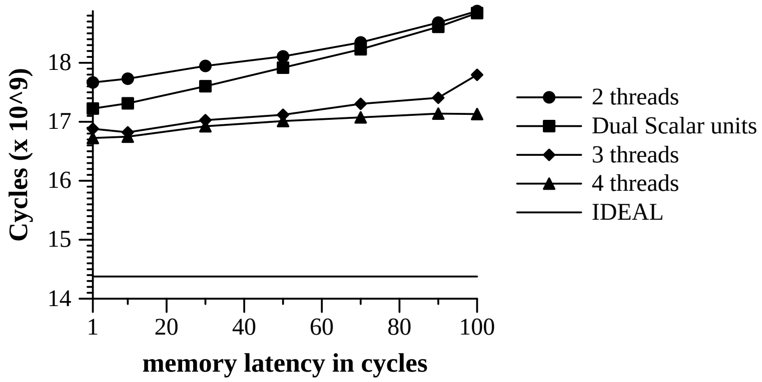

Figure 7.9 Comparison between a single multithreaded control unit and 2 control units sharing the vector facility.

scheduling is biased towards benefiting thread 0, this change in position results in a slightly lower execution time.

7.7 MULTITHREADING VERSUS REPLICATING THE SCALAR PROCESSOR

There has been a commercial machine – the Fujitsu VP2000 series [Uch91]– featuring a vector processor being shared by two scalar units. This family of vector processors has a configuration where one vector processor can be shared by two scalar processors (termed Dual Scalar Processing in their terminology [Uch91]). This is similar to having a multithreaded machine of 2 contexts but with the advantage of having two scalar units fetching, decoding and executing in parallel.

In this section we compare the performance of a machine having two replicated scalar units each of which has the same power as the scalar unit of the reference architecture versus the fully multithreaded approach of 2 contexts we have been using. Thus, the main difference between the two approaches is that the Fujitsu-style machine can issue up to 2 instructions per cycle, while the multithreaded machine with two contexts is still limited to issuing 1 instruction per cycle. We will also reproduce, for the sake of comparison, the performance of the multithreaded machines with 3 and 4 contexts.

Figure 7.9 presents the comparison of the execution time of the two schemes for several values of memory latency. What can be seen from this figure is that, as expected, the Fujitsu-style machine has a better performance than the pure multithreaded. This is due to the ability of the Fujitsu-style machine of decoding and executing two scalar instruction per cycle. At low latencies, when memory responds very fast, this "superscalar" effect shows and gives the Fujitsu-style machine an advantage of almost 3%. The advantage is not very large because the amount of scalar code in our benchmarks programs is relatively low (see table 3.1). When latency increases, the critical path of the programs is enlarged and the relative weight of the scalar code over the total execution time diminishes. Therefore, at large latencies the positive effect of having two scalar units is diminished. This is the main reason why the two curves for the Fujitsu-style and 2-contexts multithreading almost converge at latency 100 (less than a 0.1% difference). The multithreaded machines with 3 and 4 contexts, as expected, outperform both other schemes.

7.8 RELATED WORK

Multithreading for scalar programs has received much attention in recent years [ALKK90, Aga92, TEE+95, TEL96, HKN+92, EJK+96] and has been found to be generally useful. The latency properties of multithreading have been asserted by several researchers [LB96, BR92].

Research has produced many alternative multithreaded designs [GHG+91, GNL95, GB96], most focusing on extending high-performance RISC cores with extra instructions or synchronization primitives to exploit thread-level parallelism. Designs combine several degrees of hardware and software cooperation to detect, schedule and execute threads from applications [LC95, TE94, KD92, FD95].

There are several ongoing projects that have built (or are in the process of building) several prototypes. The *T machine being developed at MIT [NPA92] is a multithreaded massively parallel architecture built around a commodity microprocessor with extra register files and special thread fork and join instructions added to it. The APRIL processor architecture [ALKK90], also from MIT, features a custom RISC design extended with special mechanisms for multiprocessing. The TERA machine [ACC+90] is a commercial attempt at producing a large scale multithreaded machine: it's main design goals are a high clock and large scalability up to 256 processors, The work presented here diverges from previous work in three key aspects. First, we study multithreading in the context of vector architectures and highly vectorizable programs. There has been some research on interleaving of vector instructions at the operation level [Tzi91, HMHY93] and there has been a commercial machine – the Fujitsu VP2000 series [Uch91]– featuring a vector processor being shared by two scalar units. Nonetheless, to the best of our knowledge, there are no published studies asserting the benefits of multithreading a vector machine from a general point of view and showing the implications of latency tolerance in the vector architecture design.

A second main difference with previous multithreaded superscalar work is that very simple hardware is enough to extract a high level of performance from a multithreaded vector architecture. Thanks to the large granularity of vector operations, we do not need sophisticated fetch and dispatch units, neither we need a high bandwidth I-cache. Moreover, the decoding hardware in the proposed machine has the same cost as in a non-multithreaded machine. Our proposed architecture has no out of order issue and no renaming features currently found in many proposals of multithreaded superscalar machines [TEL96, GB96]. This simple design is possible due to the fact that the control unit of a vector processor is typically idle for many cycles [EVP+95] and thus can be shared by several threads without penalizing any of them. We have focussed on the simplest design that could achieve a close to optimal level of performance.

As a third point, and also a consequence of the large granularity of vector operations, we have seen how a very small number of threads is needed on a multithreaded vector machine with a single port to almost saturate this resource.

Finally, we will also note that the type of thread scheduling used is targeted towards favoring chaining between vector instructions as much as possible. This is in contrast with typical round robin (and variations) policies found for multithreaded superscalar machines.

Regarding the possibility of combining multithreading and other latency tolerant techniques, there have been two proposals in the scalar processor community. Susan Egger's group has developed what they term a "simultaneous multithreading" processor, that combines out-of-order execution, register renaming and multithreading. By using an out-of-order engine with renaming they can easily mix computations from different threads inside the cpu. The only requirement is a fetch engine that can fetch from different threads simultaneously [TEE+95, TEL96]. Dorojevets [DO95] has introduced a mix of the decoupling and multithreading concept, as already mentioned in section 5.5.

7.9 SUMMARY AND FUTURE WORK

In this chapter we have presented multithreaded vector architectures. We have described a basic multithreaded vector architecture that with very simple hardware and a small number of threads can achieve large speedups. We have presented data showing that 2 threads sharing a single decode unit (a maximum issue rate of 1 instruction per cycle) can achieve speedups ranging from 1.2 up to 1.5.

We have also shown that the multithreading technique can almost saturate the memory port, the most scarce resource in the reference architecture. Data presented showed that with just 2 threads it is possible to achieve around an 85-90% utilization of the memory port. Increasing the hardware to support 3 threads, leads to an utilization of the memory port of around 90-95%.

The granularity of vector instructions makes the task of multithreading a vector machine relatively easier than the task of multithreading a superscalar machine. For example, the architecture presented has only one decode unit, able to look at one and only one instruction every cycle. This is in contrast with the sophisticated issue/dispatch units presented in the multithreaded superscalar literature. Moreover, we do not use out of order techniques nor we use register renaming in our proposal.

Nonetheless, the multithreading proposed does not come without a cost. We have looked at the impact of increasing the vector register file size and found that most probably the larger register file would require an extra cycle to be read and written. Fortunately, the impact of these extra cycles has been found to be less than 1%. The small impact is explained by the length of the vector instructions, by the multithreading technique and by the underlying chaining performed by the vector machine.

We have also looked at the impact of memory latency on the multithreaded machine. As expected, multithreading tolerates relatively well latencies ranging from 1 cycle up to 100 cycle, with the difference between the two extremes being around a 6%. This fact has profound implications in the design of future vector architectures. First, it shows that a very high performance memory system is not necessarily needed in a vector machine if the base architecture supports multithreading. This means that currently very expensive SRAM memory systems could be substituted by slower and cheaper DRAM memory modules, greatly reducing overall cost.

This chapter has presented a simple approach to multithreading in vector architectures. We are currently working on several aspects to still improve occupation of the memory port: using a decode unit capable of dispatching instructions from several threads, introducing register renaming and fine tuning the scheduling policy between threads. We are also working on an extension of this work for Cray-like machines having 3 memory ports. This types of machines, due to their higher parallelism, will require simultaneous issue of instructions from different threads and better scheduling policies in order to also saturate its memory ports while keeping the number of threads reasonably low.

8

A VICTIM CACHE FOR VECTOR REGISTERS

Summary

In this chapter we introduce the concept of a victim cache for the register level of a vector processor. This small cache holds full vectors "evicted" from the register level through store instructions. The target of this cache is to filter out the abundant spill code present in vector programs. The technique presented is a generalized version of the particular dynamic load/store elimination techniques we have seen in chapters 5 and 6. The victim cache eliminates around a 5–20% of load spill traffic, and provides speedups in the range 1.06–1.20. In order to allow elimination of store spill traffic, we also present a write-back victim cache that increases the amount of traffic eliminated, up to the 10–50% range.

8.1 INTRODUCTION

Spill code is a serious problem on architectures that have a limited number of registers at the ISA level. The small number of logical registers available to the programmer or compiler induce extra traffic between the processor and the next memory level in four different ways. First, when the compiler runs out of registers it must insert a store instruction to spill some register to the stack and, later, insert a load instruction to reload the data. Second, a register may be loaded from memory, but the data can not be saved for long because the register is needed for some other instruction. Then, when the memory data is needed again, it must be re-loaded. Third, variables that are reused between close iterations of a given loop are difficult to keep in the register file due to the high register pressure. Optimizations such as scalar replacement [CCK90] are hardly useful when the compiler has so few logical registers to use. Fourth, call return sequences, typically to intrinsic math routines, might involve saving and restoring of vector registers.

In the context of traditional vector architectures, it is typical to find only a few vector registers available to the programmer. The successive Cray machines [Rus78, Fat89] and mini supercomputers such as Convex C3-Series [Con92] and Alliant [PM86] only have 8 logical vector registers. In this type of machines spill code can be very detrimental to performance due to two main reasons: first, when a vector register is spilled it involves a very large amount of data movement, and, second, the distance, in terms of processor cycles, between a vector register and main memory can be rather large.

Moreover, vector processors are generally found in multiprocessor systems. On such types of machines, an expensive interconnection network between the processors and a highly interleaved main memory system is needed. The more traffic the network has to support, the more likely it is that network degradation occurs. Every bit of traffic added to the memory system due to spill code problems adds to the average memory latency seen by each individual vector processor.

All in all, the limited number of logical registers available to the compiler can severely hurt performance of an individual program and also diminish total system throughput by putting more traffic on the memory system that would actually be necessary.

The obvious solution to this register limitation is to add more physical and logical registers to a given machine. Alas, such solution is not always feasible since binary compatibility if often more important than an increase in performance. Also, instruction sets that were not designed with an extension of the vector register file in mind might have opcode space problems to specify the extra registers. Even if the ISA extension is possible, old, dusty-deck codes might require a recompilation step to benefit from these extra logical registers, and this recompilation might not always be possible.

In chapters 5 and 6 we have seen two different dynamic load/store elimination techniques that took advantage of the particular features of each architecture. In the DVA, the input/output queues were used to hold spill code and the disambiguation hardware was able to both bypass loads data and eliminate stores. In the OOOVA, a tag structure was used to detect exact matchings between loads and stores and between loads and loads. When a match was found, the rename table was artificially altered to perform a pseudo-data copy in essentially 0 cycles.

In this chapter we present a general technique to dynamically eliminate the excess traffic caused directly or indirectly by the limited number of vector registers available. The technique is more general than the two techniques already seen for the DVA and OOOVA in the sense that it does not require any special feature in the architecture. The proposal is based on the concept of a victim cache [Jou90] next to the vector register file, and thus it could be added to any of the four architectures that have been studied in this thesis: the reference machine, the DVA, the OOOVA or the MVA. For simplicity of analysis we have chosen to apply the victim cache for registers to the reference machine.

As we did for the two load elimination techniques already presented, our goal is to present a solution that is fully binary compatible with existing vector instruction sets. Thus, our technique is fully transparent to the end user and does not require recompilation nor the addition of special opcodes to the ISA.

As it will be seen, the victim cache is not only useful for reducing spill code traffic, but it can also detect and eliminate normal, non-spill traffic. Some vectorizable codes do have temporal locality between stores and loads that are close enough (maybe just one iteration away from each other) to allow hits in the victim cache.

8.2 SPILL CODE IN VECTOR PROGRAMS

In section 3.5, page 44, we already presented an analysis of the amount of spill code present in our programs. We reproduce here, for ease of readability, table 3.4. Table 8.1 presents

Program	Total Traffic	Load Spill %	Store Spill %	Total Spill %	Avg. reuse
`swm256`	4498.5	7.0	7.0	14.0	1.0
`hydro2d`	1770.7	1.2	1.2	2.4	1.0
`arc2d`	1929.9	6.3	4.5	10.8	1.4
`flo52`	691.4	5.9	5.9	11.9	1.0
`nasa7`	1721.6	1.2	1.2	2.4	1.0
`su2cor`	1494.8	13.5	6.9	20.3	2.0
`tomcatv`	513.5	20.3	20.3	40.5	1.0
`bdna`	700.6	38.0	31.5	69.5	1.2
`trfd`	657.3	0.0	0.0	0.0	–
`dyfesm`	397.9	0.1	0.1	0.2	1.0

Table 8.1 Percentage of spill code in vector memory operations. Column 2 is in millions of operations.

again the percentage of spill present in each benchmark. Refer to page 45 for an explanation of the table fields.

Looking at the data presented in table 8.1 we can see that the potential improvements in total traffic that could be achieved by the victim cache are quite high.

8.3 VICTIM CACHE FOR REGISTERS

Our proposal for eliminating excess traffic due to spill is based on the traditional concept of adding a small cache inside the processor that will hold some vector data (see Fig. 8.1). This small cache will *only* hold store traffic and this store data will be kept in the cache for as long as possible. Vector loads will check if the data they are loading is in the cache. If so, they will read it from there instead of accessing main memory.

It is very important to remark that load data is never kept in the cache. Each slot in the cache will be able to hold a full 128-element vector and, hence, the total size in slots of the cache will necessarily be small (on the order of 4-16 slots). If load data were to be kept in the cache it would rapidly overflow its contents and negate any of the spill code elimination effects that we are seeking. This is the main difference of this proposal with respect to previous cache proposal for vector processors, such as [KSF+94, Yan93, GS92, FP91b].

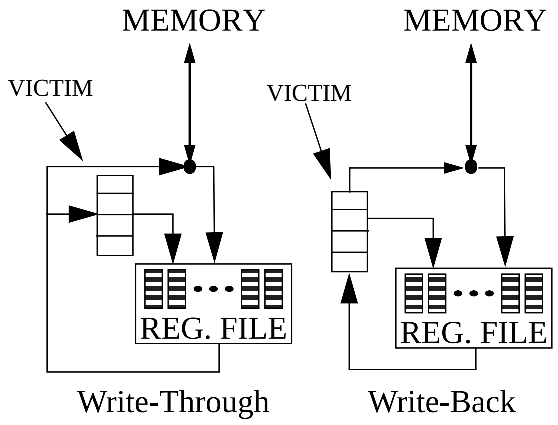

Figure 8.1 Write-Through and Write-back architectures.

We term this cache as a victim cache because it only keeps data that has been “evicted” form the register file through a store instruction. In this sense, our cache follows the same spirit as the victim caches proposed and studied in [Jou90]. The main differences with these other approaches is that our victim cache is right next to the register level, instead of next to a first-level cache.

Depending on the write policy of the cache, either write-back or write-through, we have slightly different connectivity patterns between the register file, victim cache and main memory. A load can be reading it while a store is updating it provided there is no memory dependences between the two.

In the write-through case, note that the path for writing into the victim is shared with the path for writing into main memory. On the other hand, we have added a full new path for reading from the victim into the vector register file. Hence, both a read from the cache and a read from memory can proceed simultaneously, or, alternatively, a write into the victim (and main memory simultaneously) and a read from the victim can proceed in parallel.

For the write-back cache, the peak parallelism is higher. The paths for writing into the victim and the path for writing into main memory are split, so we could have up to three memory operations proceeding in parallel: a store that does not imply an eviction from the victim cache, a load from the victim (without conflicts with the store) and a load from main memory (ditto). In the much more frequent case that a store operation forces the eviction of some slot of the cache, the parallelism drops to 2: the store (plus the eviction) and a load from the victim cache.

Each slot in the victim cache holds a full vector. Therefore, the tag associated with each slot in the cache can not be a simple memory address. Instead, we use the concept of a *region of addresses* to describe the contents of each slot, as already described in chapters 5 and 6. Each tag is a 6-tuple: $\langle @_1, @_2, \mathtt{vl}, \mathtt{vs}, \mathtt{sz}, \mathtt{v}\rangle$. Virtual addresses $@_1$ and $@_2$ define a consecutive region of bytes in memory and vl, vs, and sz are the vector length, vector stride and access granularity used when the tag was created; v is a validity bit. The address $@_2$ is computed as $@_1 + (vl - 1) \times vs + sz$.

Regarding scalar data, the cache works as follows. In the write-through case, scalar loads simply ignore the cache, while in the write-back case, if a scalar load has a conflict with a slot of the cache, the slot is sent to memory prior to performing the load. Scalar stores are disambiguated against all tags present in the cache. If an overlap between the region of addresses defined by the scalar store and any of the region of addresses found in the cache is detected, then the slots that overlap must be updated: in the write-through case, we simply invalidate them. In the write-back case, the slots must be sent to memory prior to performing the store.

8.4 EXAMPLES OF TRAFFIC ELIMINATION

We re-take now the spill code examples already presented in section 3.5.1, page 49. We will show using the `su2cor` example how the victim cache can eliminate spill code. We use a a particular version of the victim cache to clarify the mechanisms for spill elimination. Later sections will look in more detail to other victim cache alternatives.

Vector Load Elimination

To see how spill code traffic can be effectively killed, consider the two basic blocks from program `su2cor` shown in Fig. 8.2. For each basic block we present a schematic view of all spill-loads, spill-stores and stores (normal loads have been removed for clarity). For each spill-load there is a back arrow connecting it to the store from where it is reading its data.

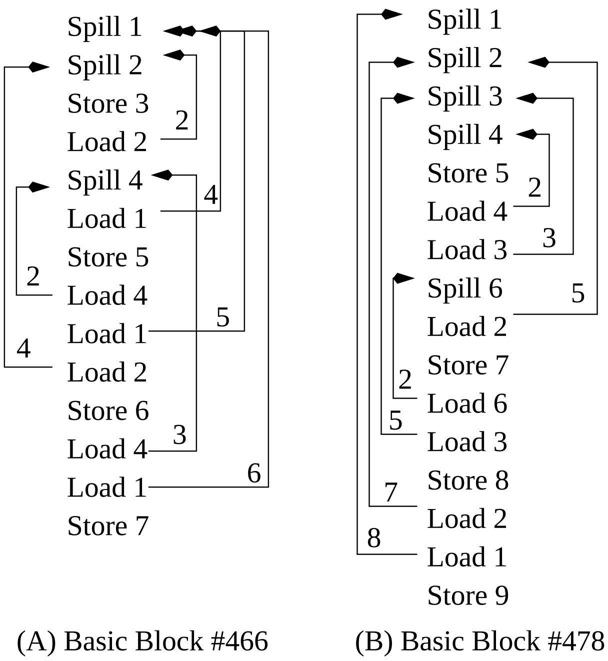

Figure 8.2 Schematic view of basic blocks #466 and #478 from `su2cor`. Included are only spill-loads, spill-stores, and normal stores. All scalar code and vector arithmetic instructions have been removed for clarity. The label on each arc is the number of intervening stores between a spill-store and its matching spill load.

We have labeled each arrow with the number of intervening stores between the spill-store saving a register and the spill-load reading it. For example, in basic block #478, between spill-store-1 and spill-load-1 there are a total of 8 stores (spill load 1 itself, spills 2, 3, 4 and 6 and stores 5, 7 and 8).

Now, assume that we add a small cache with FIFO replacement that is able to hold, say, 4 vector registers (we will use the abbreviation "FIFO cache" to refer to a cache with FIFO replacement). Each time a vector store is performed, a copy of the data is kept in this small cache. If the cache is full, the first store is discarded and in the slot being freed the new store is copied. In our example, after executing the normal-store #3 we would have three full slots in the cache. When load-2 is performed, it checks in the cache. Since it finds the data it was looking for, it uses the special data-path to copy the data from the FIFO cache into the destination register. The memory port is not used at all and is available for a subsequent load instruction.

By merging and ordering all the labels on each arc found in the two basic blocks of Fig. 8.2 we obtain the following sequence: 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 5, 6, 7, 8. From this ordering we can statically determine that with a FIFO cache-like structure of 4 slots we should be able to catch 8 spill-loads and enlarging the cache up to 8 slots would catch all spill-loads for these two blocks.

Vector Store Elimination

The possibility of dynamically store elimination can also be considered. Instead of having a write-through policy for this FIFO cache just described, we could use a write-back policy. When a store is first performed, it is always kept in the FIFO cache and does not update memory. Only when the store is evicted from the cache, memory is actually updated.

Every vector store performed must be checked against all stores already present in the cache. If there is overlap between the younger store and one or more of the older stores, these must be sent to memory and then, their slots in the cache invalidated. If there is an exact match between the young store and one of the older stores, then this older store can be invalidated without sending it to memory, since the most up-to-data data for that particular memory region belongs to the younger store.

The problem with dynamically eliminating a vector store is that main memory is not in a precise state if an exception happened to occur. Since we have completely discarded the old store, we can not precisely recover the memory state if the exception occurred between the old and the young store. This may or may not be a problem depending on the requirements of the baseline architecture. If precise exceptions are required, then store traffic can not be eliminated.

Going back to our example of Fig. 8.2, we can statically determine the size of the victim cache needed to eliminate store spill traffic. Consider store-spill-1 of basic block #466. In iteration i this store data is inserted in the victim cache, and main memory is *not* updated. If we reach iteration $i+1$ and store-spill-1 is still in the victim cache, when we re-execute the same store instruction it will have an exact hit against the old store-spill-1. This old store will be removed from the cache without generating any extra traffic and the new store will be inserted in the victim cache. For this situation to happen, the victim cache needs to be able to hold 7 different vectors. With 7 slots and a FIFO replacement policy, all three spill stores present in loop #466 would be eliminated. If the loop performs N iterations, only the spill stores from the last iteration might actually go to main memory, with all intermediate spills being eliminated by the victim cache. Similarly, for loop #478, 9 slots in the victim cache would be needed to eliminate the 5 spill stores.

8.5 VICTIM CACHE PARAMETERS

The victim cache is a small fully associative cache. We have studied caches having 4, 8 and 16 slots. In all cases, the cache behaves as write-allocate: that is, a vector store always requires a slot in the victim cache.

Regarding the replacement policy we have looked at three variations. The first policy under study is an LRU replacement policy, where the least recently referenced slot is the one being replaced. In case one or more slots are empty or invalid, then one of these slots is preferred for replacement rather than using whatever slot the LRU algorithm indicates. The second policy is a pure FIFO replacement, ignoring empty and valid bits. A cyclic counter indicates what slot is the next one to be replaced. The counter is incremented each time a new store enters the cache. Although the FIFO scheme sounds odd for a cache, it is well suited for spill code. Looking at the two basic blocks presented in Fig. 8.2 we can see that there is a certain degree of locality of references to nearby stores.

The third policy, termed FIFO+MRR, is a potential enhancement of the FIFO just described. Looking back to table 8.1 we can see that a vector that is spilled is most often re-loaded only once. Looking at column labeled "avg. reuse" we can see that most programs spill and re-load each register only once. To take advantage of this behavior, we add a reference bit to each slot. When looking for a slot to be replaced, we first check for invalid slots, empty slots or slots that have already been referenced (their reference bit is 1). If one of these is found, we use it for replacement. If not, the FIFO ordering dictates which slot must be replaced.

8.6 PERFORMANCE RESULTS

8.6.1 Write-Through Victim Cache

We will consider the performance of the write-through victim cache under two different models. In the first model, only the vector store traffic properly identified as spill code is cached

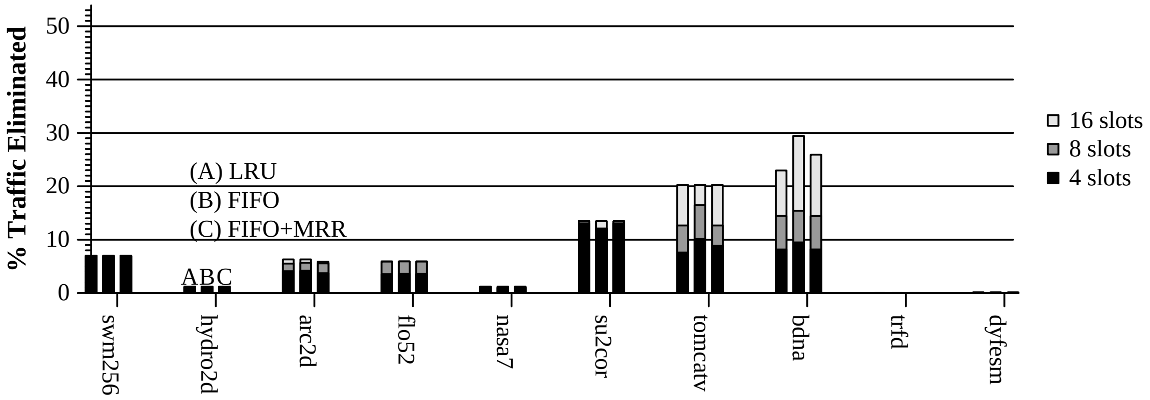

Figure 8.3 Percentage of total traffic eliminated by the Write-through victim cache under the SPILL-ONLY model.

in the victim slots (model SPILL-ONLY). In the second model, all store traffic, both normal and spill stores, are cached in the victim slots (model ALL-STORE).

How do we identify and differentiate stores that are spill code from normal stores at run time ? There are several alternatives. In this work we do it by monitoring vector stores that use register `a7` for their base address calculation. Since the compiler always stores spilled registers in the stack and all accesses to the stack are made relative to `a7`, it is straightforward to distinguish normal and spill stores. This scheme has the drawback of relying on the compiler being used. If, for whatever reason, the compiler changed the spilling scheme, the victim cache would be no longer able to distinguish stores.

In the following sections we will look at the performance of the victim cache from two related points of view: percentage of traffic served from the victim cache and speedup of each program with the victim cache relative to the reference machine.

Traffic Reduction

Figures 8.3 and 8.4 present, for each program, the percentage of data elements that were retrieved from the victim cache and that did not access main memory. This percentage is over total vector and scalar memory traffic Fig. 8.3 shows the traffic served from the victim cache under the SPILL-ONLY model while Fig. 8.4 presents the same data under the ALL-STORE model.

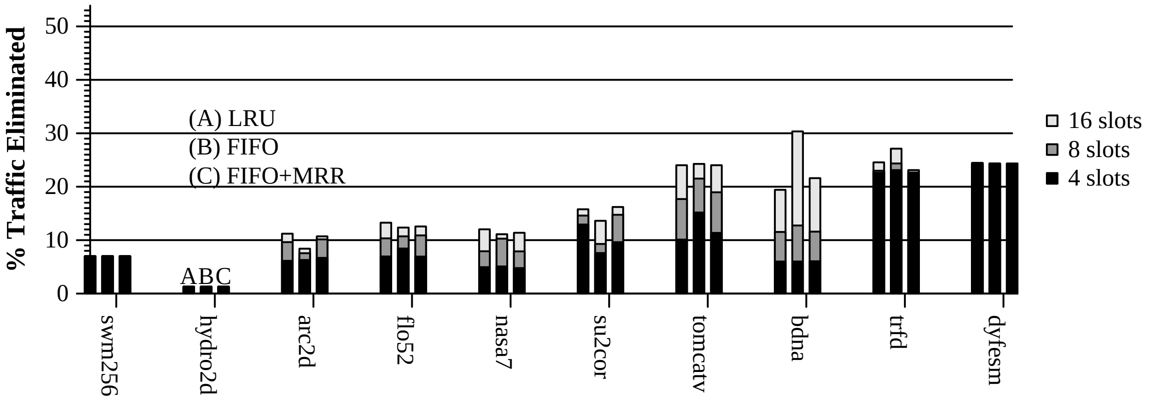

Figure 8.4 Percentage of total traffic eliminated by the Write-through victim cache under the ALL-STORE model.

For each program we show three different bars, one for each of the replacement policies described above (LRU, FIFO and FIFO+MRR). Also, for each policy we present data for three caches having 4, 8 and 16 slots respectively.

Note that we present this SPILL-ONLY model mostly for evaluation purposes. We believe the ALL-STORE model would be the implementation of choice in most real cases. The rationale for studying both models is to be able to show the fraction of normal store traffic captured by our proposal.

When the victim cache is used to hold spill only data, the amount of load traffic served from the cache is very high in relative terms. The absolute traffic reductions vary between 0 and 4% for four programs, between 5 and 10% in three programs (`swm256`, `arc2d` and `flo52`) and are between 13 and 20% for `su2cor`, `tomcatv` and `bdna`. Yet, comparing the percentages shown in Fig. 8.3 to the third column of table 8.1 (percentage of spill-loads) we see that just 4 slots seem to be able to capture almost all available spill load traffic. For example, `swm256` has a 7% of spill load traffic and the victim cache captures a 7% of load traffic. For program `arc2d`, with a 6.3% of spill loads, the victim cache captures a 4% of all load traffic. In only two cases, `tomcatv` and `bdna`, the cache falls short of its full potential: for `tomcatv` it only captures a 10% of all load traffic while the spill load traffic represents a 20%. For `bdna`, the cache captures a 9.5% of traffic while it could theoretically be able to capture 38% of all load traffic.

To be able to capture the remaining load spill one needs to enlarge the cache up to 16 slots. As it can be seen, both `tomcatv` and `bdna` almost reach their peaks (20.3% and 38%) for the 16 slots victim cache.

As for the effectiveness of each replacement policy, we can see that the differences are very small when the slots are only 4 or 8. Only at 16 slots there is a peak favoring the FIFO policy for program `bdna`. Since the cache has very few slots (albeit each of them is big) the cost of one policy versus another is very small. Since the FIFO policy is simple and seems to yield slightly better performance it would be our choice.

Comparing Fig. 8.4 to Fig. 8.3 we can see that there is a fair amount of "hidden" spill code in our set of benchmarks. Except for `swm256` and `hydro2d`, the other 8 programs clearly increase the amount of traffic served from the victim cache when *all* stores are walked through the cache. In the case of `trfd` and `dyfesm`, the rise is very spectacular, reaching 30% of all load traffic. In the case of `bdna`, though, the fact that normal, non-spill, traffic is going through the cache decreases the percentage of traffic served for the LRU and FIFO+MRR cases.

Our analysis of `tomcatv` and `dyfesm` show that all this traffic suddenly captured by the victim cache comes from (a) recurrences and (b) unsolved intra-iteration dependences that the compiler could not rearrange. We refer the reader back to the analysis we already performed for `trfd` in section 5.4.7, page 115.

Speedup

Figures 8.5 and 8.6 present the associated speedups achieved by the victim cache under the same assumptions described in the previous section.

For the SPILL-ONLY model, typical speedups at 4 slots range between 1.02 and 1.05 with only `tomcatv` and `bdna` slightly exceeding 1.05. On the other hand, under the ALL-STORE model typical speedups range between 1.05 and 1.10, with a couple of programs reaching speedups over 1.15 (`tomcatv` and `bdna` at 16 slots).

The main question is why all the traffic reduction seen in the previous section does not translate in higher speedups. When a vector load is serviced from the victim cache the performance benefits are threefold: first, the latency delay is only 1 cycle (in contrast with 50 cycles of main memory latency). Second, chaining is allowed between the load that hits

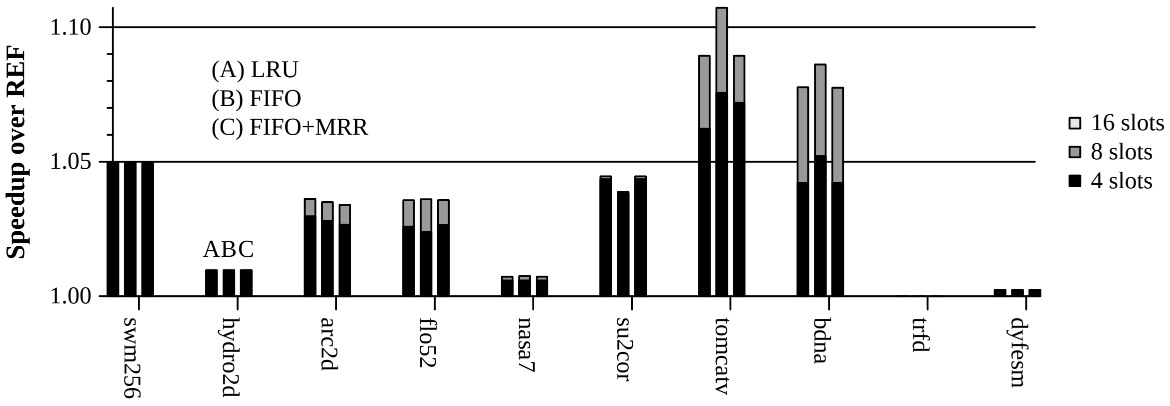

Figure 8.5 Speedup due to the Write-through victim cache under the SPILL-ONLY model.

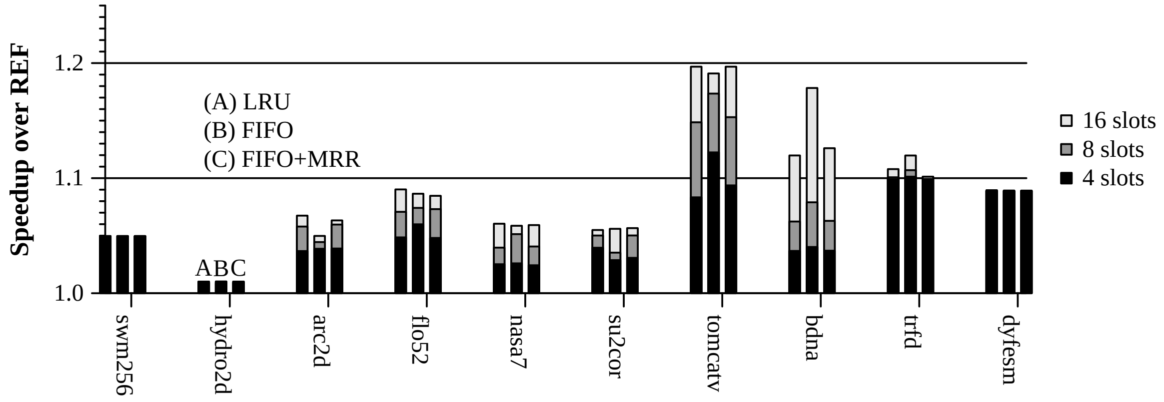

Figure 8.6 Speedup due to the Write-through victim cache under the ALL-STORE model.

in the cache and a subsequent operation using the register being loaded. Third, and most important, the memory port is idle and could and should be used to immediately start another load operation.

The problem is that this third advantage is difficult to achieve. Whenever a load that reads from the victim is dispatched, there is a high probability that the decoder will stall within

Program	FIFO		
	4	8	16
swm256	1.3	1.4	1.4
hydro2d	0.0	0.1	0.2
arc2d	1.9	2.0	2.2
flo52	1.3	1.5	2.1
nasa7	1.8	3.8	3.9
su2cor	2.0	2.4	3.6
tomcatv	8.4	11.8	12.2
bdna	2.7	4.4	10.2
trfd	10.2	10.8	12.1
dyfesm	8.7	8.7	8.7

Table 8.2 Percentage of total execution cycles where the memory port and the victim cache port were both simultaneously busy.

two or three instructions before seeing another load that could use the main memory bus. This behavior is due to the in-order simplicity of the underlying vector processor. The decode unit only fetches one instruction per cycle and does not do any instruction look-ahead.

We have measured the percentage of cycles where the vector processor manages to use at the same time the main memory port and the victim cache port. Table 8.2 presents these percentages for the FIFO case and for the three cache sizes. As it can be seen, in most programs the fraction of cycles where the full bandwidth is used is very low. Only for three programs is close to 10%. When the cache size is increased, the percentages also increase, but still never exceeds 12%.

The conclusion from this data is that the victim cache is acting mostly as a latency elimination technique, without delivering an effective bandwidth improvement to the processor. Although the victim cache adds a second input port to the register file, the data shows that the main memory port and the victim port are seldom used together. If the processor had some out-of-order techniques, such as the ones described in chapter 6, the chance for using both ports simultaneously would increase.

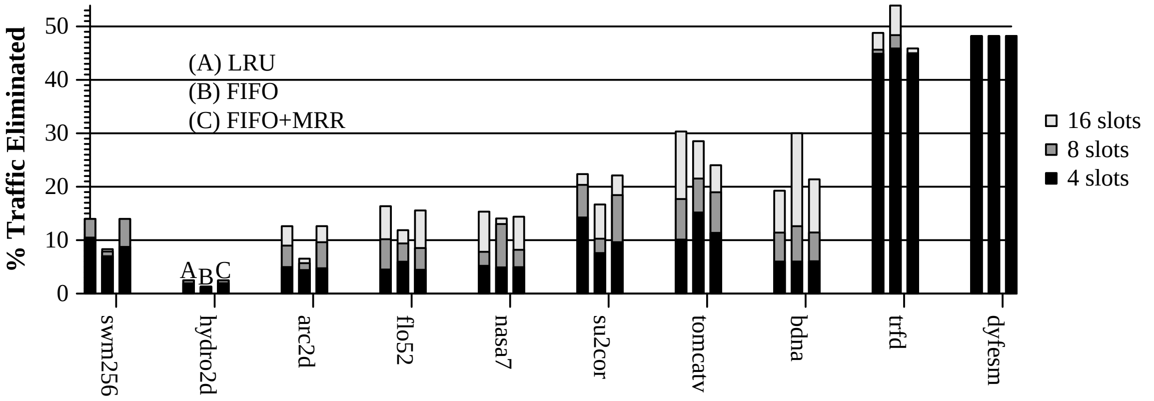

Figure 8.7 Percentage of total traffic eliminated by the Write-back victim cache (ALL-STORE model).

8.6.2 Write-Back Victim Cache

In the write-back victim cache all stores must go through the victim cache. Therefore, the ALL-STORE models will be the only under study in the following sections.

Traffic

Figure 8.7 presents the fraction of traffic eliminated when using a write back cache, following the same conventions as described in section 8.6.1 for Fig. 8.4.

In almost all programs, the fraction of traffic eliminated is increased with respect to the traffic for the write-through ALL-STORE model presented in Fig. 8.4. Programs `trfd` and `dyfesm` have an almost twofold increase in traffic eliminated, due to their special characteristics already mentioned.

Comparing the results to column "Total" of table 8.1, we see that the victim cache is capturing more traffic than the spill present in the programs (except for `tomcatv` and `bdna`, where the spill traffic amounts to 40 and 70% and the victim only eliminates a 30%). This is especially notorious in `nasa7`, `trfd` and `dyfesm`, where the traffic being eliminated is not pure spill code, but is the result of a caching effect.

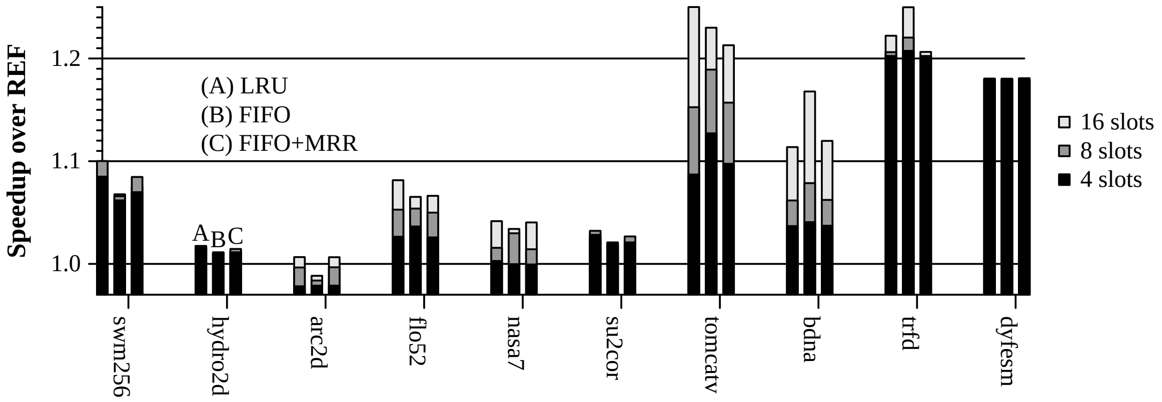

Figure 8.8 Speedup due to the Write-back victim cache (ALL-STORE model).

Speedup

Figure 8.8 presents the speedups over the reference machine when the write-back victim cache is used. Surprisingly enough, although the amount of traffic eliminated by the write-back victim was larger than the traffic eliminated by the write-through victim, 5 programs present worse speedups in the write-back case than in the write-through case. Programs `arc2d`, `flo52`, `nasa7`, `su2cor` and `bdna` all achieve lower speedups over the reference machine that the ones presented in Fig. 8.6. Program `arc2d` presents an even worse situation, since at 4 slots, the program takes longer to execute with the write-back cache than with nothing at all.

Programs `swm256`, `tomcatv`, `trfd` and `dyfesm` behave as expected, and their speedups are parallel to their increase in traffic eliminated.

What is the performance penalty being introduced by the write-back victim cache ? An analysis of the program reveals that these lower speedups are the result of invalidations in the victim produced by load operations. Since we have a write-back cache, when a load is executed main memory is not (necessarily) up to date. Therefore, a load operation must be disambiguated for consistency with all stores present in the victim. If the load is found to overlap with one or more of those stores, the victim slots involved must be sent to memory prior to performing the load.

Program	LRU			
	4		16	
	%	cost	%	cost
swm256	n.a.	n.a.	n.a.	n.a.
hydro2d	0.3	85.9	0.8	86.8
arc2d	12.8	95.1	13.1	94.7
flo52	13.3	24.3	14.4	26.6
nasa7	8.4	71.5	10.9	61.2
su2cor	11.7	123.3	n.a.	n.a.
tomcatv	0.0	0.0	n.a.	n.a.
bdna	5.2	30.9	7.4	50.2
trfd	0.7	13.9	1.0	16.5
dyfesm	4.4	15.4	7.4	12.5

Table 8.3 Invalidations in the write back cache: column '%' indicates percentage of all loads that caused an invalidation in the victim. Column 'cost' indicates the average number of data elements sent to memory at each invalidation.

When this situation happens, all the time needed to send the victim slots to memory adds up to the effective latency seen by the processor for that particular load. Since loads are almost always in the critical path of our programs, the total execution time is likely increased by the number of cycles taken to put memory in an up-to-date state. For example, if one slot of the victim cache containing a 95 element store must be sent to memory prior to performing a certain load, then the effective latency of that load will be the 95 cycles required to evict the store plus the main memory latency (50 cycles) corresponding to the load.

We measured for each program two indicators of this situation: (a) the percentage of loads over all vector loads that caused an invalidation and (b) the average cost (in cycles) of each invalidation. The cost of an invalidation is taken as the number of cycles needed to evict all the slots that have a conflict with a load. The results are presented in table 8.3. Column '%' indicates the percentage of all loads that caused an invalidation and column 'cost' indicates the average number of data elements evicted from the victim for each invalidation. We present the data only for the LRU case with 4 and 16 slots due to lack of space. The data for all other cases is very similar.

Table 8.3 shows that in all five programs where the unexpected decrease in speedup occurred there is a large percentage of loads that cause an invalidation (> 10%) and that the average cost of an invalidation was also large. On the other hand, program `bdna`, which also shows some degradation but in a lesser degree, has only a 5% of loads causing invalidations and each invalidation was relatively cheap (only around 30 elements being evicted from the victim).

The general conclusion is that the write back cache provides better traffic elimination than the write through but that, due to invalidation costs, it actually performs worse. We are currently studying an hybrid approach where spill stores are write-back while normal stores are write-through.

8.7 EFFECTS OF MAIN MEMORY LATENCY

We finally present a study of the effects that different memory latencies have on the victim cache proposal. For the sake of brevity, we will present data referring only to our best performing configuration, that is, the write-through, all-store, FIFO victim cache. Other configurations behave in a very similar way.

Figure 8.9 presents the total execution time of each program when main memory latency is varied from 1 to 50 to 100 cycles. We present the execution time for the reference machine (curve labeled "REF") and for the write-through, all-store victim cache for three different cache sizes: 4, 8 and 16 slots (labeled "WT 4", "'WT 8" and "WT 16" resp.).

The figure shows that even being a vector processor, all programs are severely influenced by memory latency. There are two important points to note: first, at latency 1, the victim cache provides an execution time advantage in those programs that were able to use the victim port and the memory port simultaneously (programs `tomcatv`, `bdna`, `trfd` and `dyfesm`, see table 8.2). For the rest of programs, loading a vector from the victim or from main memory is almost the same, since in both cases the latency is 1 cycle. Thus, for these other programs, the speedup should be either very small or derived from the fact that loading from the victim cache allows chaining of subsequent operations.

Second, the most important effect of the victim cache is changing the slope of the execution time curves, thereby making the processor more latency-tolerant. See how, for example, the slope of the REF and WT curves for program `bdna` are clearly different. This effect would allow to trade off the cost of the memory system against the cost of the victim cache: for example, program `tomcatv` could have a slower memory system and compensate the loss of performance by adding a victim cache.

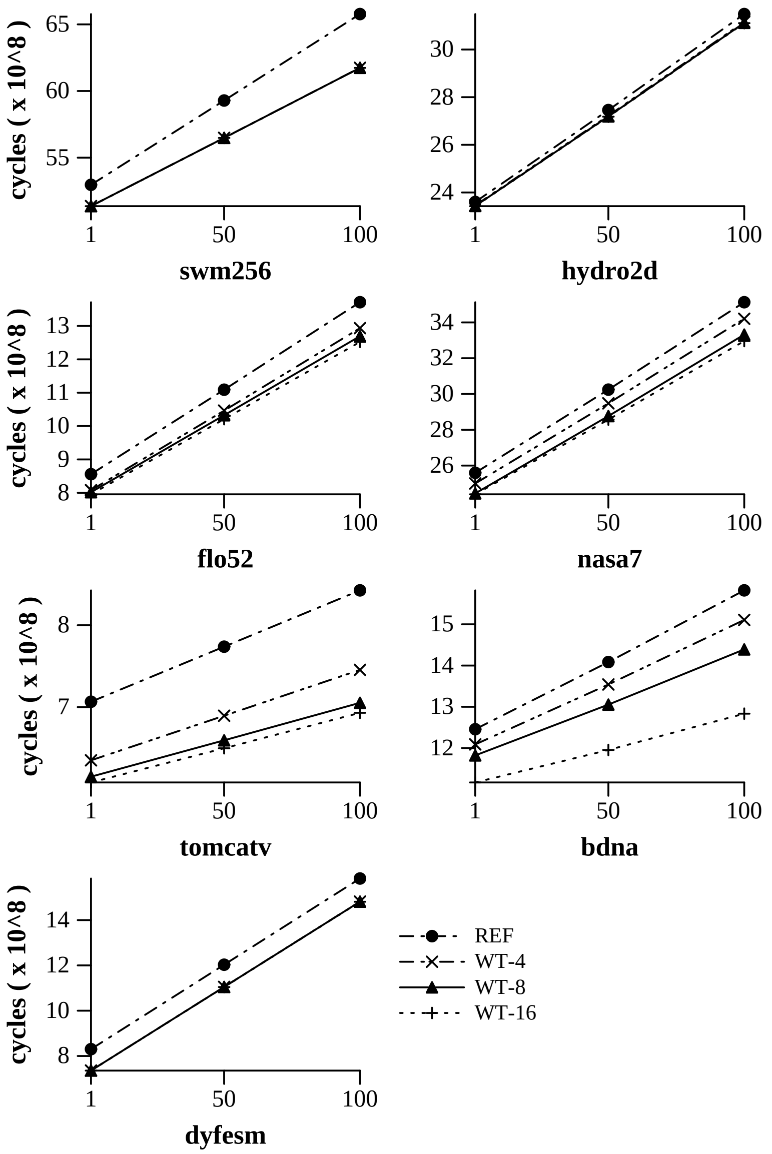

Figure 8.9 Execution time comparison of the reference architecture ('REF') and three configurations of the write-through, all-store victim cache, under different memory latencies.

8.8 RELATED WORK

This chapter has presented a dynamic spill-code elimination technique based on the concept of caching. To the best of our knowledge, this is the first general technique presented to deal with this specific problem in vector processors.

Previous work has focused on general caching techniques for all load/store traffic. For example, Fu and Patel [FP91b] studied caching for the Perfect Club programs and report that results are very much dependent on stride patterns. Gee and Smith [GS92] also studied the general problem of caching using traces from a Cray X-MP and the Ardent Titan, but focused on caching all types of references. Following the general caching technique, Yang [Yan93] presented the prime-mapped cache for vector machines, which improved upon previous designs. Kontothanassis et. al. [KSF+94] compared the performance of a DRAM main memory systems against the traditional SRAM memory system and against different sizes of cache based DRAM memory systems.

8.9 CONCLUSIONS

In this chapter we have considered the usefulness of adding a victim cache next to the register level of a vector processor. We have seen through measurement of numerical vectorizable codes that the low number of available vector registers causes the introduction of a large quantity of spill code. We have shown that the victim cache is able to filter this spill code traffic and serve most of the data requested by the processor without accessing main memory.

Comparing the victim cache technique to the two techniques seen for the DVA and MVA we can highlight the following differences. The DVA implements load/store elimination using the load/store data queues present in a decoupled vector architecture. The OOOVA uses the renaming mechanism to dynamically eliminate load traffic. In each of these two previous implementations, a particular architecture feature (queues and register renaming, resp.) was being exploited to achieve the traffic elimination. The technique presented here starts from a traditional, in-order, vector architecture and adds a victim cache to it without modifying its basic execution paradigm.

We have studied several configurations of a write-through victim cache that was able to eliminate most of the load spill traffic. With just 4 slots, simulations have shown that between a 5 and a 20% of load traffic can be eliminated.

Although the study was initially targeted at eliminating spill code traffic, when all store traffic is kept in the victim cache, we have seen that some extra traffic elimination is achievable. In particular, hidden spill and reused vectors between iterations of a loop can be captured when all the store traffic is walked through the victim cache.

The speedups associated with these traffic reductions go anywhere from 1.05 up to 1.20, with the speedups associated with the all-store model being larger than the spill-only model.

In order to also eliminate spill store traffic, we have studied a write-back version of the victim cache. The goal was to keep spill stores of a given loop long enough in the victim cache so that at each successive iteration of the loop a spill store would replace in the cache its own copy from the previous iteration of the loop. The traffic reductions associated with this write-back victim were much larger than those of the write-through cache, typically in the 10–50% range.

The write back cache, though, had two drawbacks. First, by its own nature, it does not allow for precise exceptions if stores are eliminated without updating memory. Second, although the total memory traffic generated was lower than in the write-through case, the corresponding speedups were, in five cases, worse. This degradation in speedup is due to invalidations caused by load operations. For these five programs we have seen that around 10% of all vector loads caused an invalidation that costed anywhere between 25 and 128 cycles. This invalidation cost adds up to the main memory latency and causes an effective delay larger than the expected benefit.

Finally, we have looked at the effect of the victim cache when main memory latency is varied from 1 to 100 cycles. The simulations show that the victim cache can improve execution time even with a perfect memory system having a 1 cycle latency provided that the instruction level parallelism in the program allows to use the main memory port and the victim cache port simultaneously. For larger latencies, the victim cache hides some of the main memory latency and allows the processor to be more latency tolerant. At latency 100 cycles, the simulations have shown speedups as high as 1.24.

Clearly the victim cache approach is not as powerful to tolerate large memory latencies as the out-of-order, decoupling or multithreading techniques seen so far. Yet is a very cost-effective approach that requires minimal hardware changes to an already existing architecture.

We are currently working on an hybrid model where normal stores are treated as write-through and spill stores are treated as write-back.

9

CONCLUSIONS AND FUTURE WORK

Summary

This chapter summarizes the main contributions of this thesis and presents an analysis of the results shown in each chapter. Future lines of work opened up by the work presented here are also discussed.

9.1 DYNAMIC SCHEDULING

We started this thesis asking ourselves whether highly vectorizable codes running on a vector machine achieved the peak performance provided by hardware resources. Much literature exists analyzing application performance on vector machines and showing that it is rather uncommon to approach the peak performance of a machine. This performance gap between actual and peak performance has slowly affected the success of supercomputers based on vector technology, and nowadays, their commercial continuity is not as promising as it used to be.

We identified several reasons why the success of vector architectures has been declining that can be summarized in the fact that their price/performance ratio has worsened when compared to other paradigms of supercomputing. While absolute performance of vector machines has increased, the memory systems required to keep these powerful machines fed with data are still large, complex and very expensive. Thus, when comparing vector mainframes against multiprocessors using CMOS cache-based superscalar microprocessors, the cost per megaflop clearly favors the CMOS multiprocessors. Moreover, cache based superscalar machines are perceived to be more flexible than vector architectures, since the latter require applications that are highly vectorizable. While cache based machines can not successfully execute programs with very large data sets, their success in capturing a large fraction of most application domains is unquestionable.

One of the key points in the cost and performance of vector architectures is their memory systems. Vector machines have many interleaved memory banks that provide very high bandwidths into the processor. Moreover, to keep latencies low, vector supercomputers use the best possible technology available for the memory modules. This typically consists in using the fastest form of SRAM main memory. This kind of memory, while providing excellent performance, can be the source of the majority of the cost of a machine.

From the user perspective, the cost of such high performance memory systems is not justified if it does not translate into application speed. Thus, the disparity between real and peak performance on a vector machine puts the price/performance ratio of vector supercomputers in a bad position. In order to remain viable, vector architectures must deliver sustained performance close to peak performance and reduce total system cost.

System cost reduction should start targeting the memory system, which makes most of overall machine cost. Changing from SRAM memory modules to DRAM memory modules would decrease the raw price of memory components at the expense of a large increase in processor-to-memory latency. To compensate this effect, the cpu should be able to tolerate memory latencies two to three times larger than current latencies. To achieve the performance improvement, the first step would be to understand the bottlenecks found in a program execution that cause the discrepancy between actual and peak performance. Once the bottlenecks are known, solutions might be devised.

We started looking at the actual versus maximum performance. Why do vector machines show this performance gap ? To investigate into this question, we instrumented several benchmark programs and collected statistics about their instruction profile and their behavior when run on a traditional vector architecture.

The instruction profile shows that all codes are memory bound when run on a vector architecture with two functional units and one memory port (chapter 3). Moreover, simulations of the programs on a traditional in-order vector architecture showed two very important points (see chapter 4):

- First, vector architectures can not tolerate large increases in memory latency. In all programs, the degradation in execution time is almost linear when memory latency increases.
- Second, even with a perfect memory system, that is, with a latency of 1 cycle, the distance between the simulated execution time and the minimum execution time of a program, is very significant. Thus, even in the absence of memory latency, there are factors that limit the performance of a traditional vector architecture.

Our analysis of the instruction level parallelism that the reference architecture extracts from the benchmarks shows that they spend a very high fraction of overall execution time in a completely idle state. Moreover, adding up the total idle time on the memory port, we see that it can range between a 30% and a 65% of all execution time. Even with a perfect memory system the idle time on the memory port is typically between a 20% and a 30%. Moreover, the results show that the fraction of time executing vector computations is not as high as one would expect (around 50%). We have presented data about the fraction of utilization of the vector functional units and found that only in roughly 25% of the cycles are all vector computation units busy, while there is a 25% percent of cycles where only one of the two arithmetic units is working.

Combining the results from chapters 3 and 4 we can see that these ten programs are clearly memory-bound and yet, they can have the memory port idle for up to a 50% of its total execution cycles. This is a counter-intuitive result, since the fact that the programs are memory bound would lead to conclude that its memory port would be the most saturated resource and that it would be busy for the majority of cycles.

This port idle time has a terrible effect since it goes against the latency tolerance needed to reduce memory cost. In order to be latency tolerant, an architecture must be making a continuous flow of requests to the memory system, so that the latency of each individual request is masked by overlapping on top of it other requests.

How can the machine become completely idle, simply waiting for data to arrive from memory ? If the programs are memory bound, how can it be that at some points we do not have memory instructions ready to be issued ?

The in-order nature of the reference machine forces the control unit to stall whenever there is a hazard in the computation part of the processor, even though the memory port might be idle. When a simple in-order execution model is used, the memory accessing stream and the computation stream interfere with each other and prevent achieving the peak speed. To overcome this in-order problem we have proposed and analyzed three different solutions: decoupling, out-of-order execution with register renaming and multithreading.

Decoupling

We have described a basic decoupled architecture (DVA) that uses the principles of decoupling to hide most of the memory latency seen by vector processors. The DVA architecture shows a clear speedup over the REF architecture even when memory latency is just 1 cycle. This speedup is due to the fact that the *AP* slips ahead of the *VP* and loads data in advance, so that when the *VP* needs its input operand they are (almost) always ready in the queues. Even if there is no latency in the memory system, this "slipping" produces a similar effect as a prefetching technique, with the advantage that the *AP* knows which data has to be loaded (no incorrect prefetches). Thus, the partitioning of the program into separate tasks helps in exploiting more parallelism between the *AP* and *VP* and translates into an increase in performance, even in the absence of memory latency. Moreover, as we increase latency, we see how the slopes of the curves of the execution time of the benchmarks remain fairly stable, whereas the REF architecture is much more sensitive to the increase in memory latency. Overall, speedups of the DVA over the REF machine vary in the range 1.2–1.6.

We have shown that this speed improvements can be implemented with a reasonable cost/performance tradeoff. Section 5.3.4 has shown how the length of the queues does not need to be very large to allow for the decoupling to take place. A vector load queue of four slots is enough to achieve a high fraction of the maximum performance obtainable by an infinite queue. On the other side, the vector store queue does not need to be very large. Our experiments varying the store queue length indicate that a store queue of 2 elements achieves almost the same performance as one with sixteen slots.

Out-of-order execution

We have also considered the usefulness of out-of-order execution and register renaming for vector architectures. We have shown that when out-of-order issue and register renaming are introduced, vector performance is increased. This performance advantage can be realized even when adding only a few extra physical registers to be used for renaming. Out-of-order execution is as useful in a vector processor as it is widely recognized to be in current superscalar microprocessors.

Using only 12 physical vector registers and an aggressive commit model, we have shown significant speedups over the reference machine. At a modest cost of 16 vector registers, the range of speedups was 1.24–1.72. Increasing the number of vector registers up to 64 did not lead to significant extra improvements, however.

Moreover, we have shown that large memory latencies of up to 100 cycles can be easily tolerated. The dynamic reordering of vector instructions and the disambiguation mechanisms introduced allow the memory unit to send a continuous flow of requests to the memory system. This flow is overlapped with the arrival of data and covers up main memory latency.

Multithreading

Finally, we have considered as an orthogonal solution the multithreading technique. Although we have applied multithreading to a traditional in-order architecture and shown that it can greatly improve throughput, it is worth pointing out that multithreading could be combined both with decoupling and out-of-order execution.

We have described a basic multithreaded vector architecture that with very simple hardware and a small number of threads can achieve large speedups. We have presented data showing that 2 threads sharing a single decode unit (a maximum issue rate of 1 instruction per cycle) can achieve speedups ranging from 1.2 up to 1.5. We have also shown that the multithreading technique can almost saturate the memory port, the most scarce resource in the baseline architecture. Data presented showed that with just 2 threads it is possible to achieve around an 85-90% utilization of the memory port. Increasing the hardware to support 3 threads, leads to an utilization of the memory port of around 90-95%.

Nonetheless, the multithreading proposed does not come without a cost. Increasing the vector register file size might put in jeopardy the cycle time and could induce the introduction of an extra latency cycle required to read and write from/to each register. Fortunately, the impact of these extra cycles has been found to be less than 1%. The small impact is explained by the length of the vector instructions, by the multithreading technique and by the underlying chaining performed by the vector machine.

We have also looked at the impact of memory latency on the multithreaded machine. As expected, multithreading tolerates relatively well latencies ranging from 1 cycle up to 100 cycle, with the difference between the two extremes being around a 6%. This fact has profound implications in the design of future vector architectures. First, it shows that a very high performance memory system is not necessarily needed in a vector machine if the base architecture supports multithreading. This means that currently very expensive SRAM memory systems could be substituted by slower and cheaper DRAM memory modules, greatly reducing overall cost.

Conclusion

The results presented in this thesis show that by using some form of dynamic scheduling we can achieve the two goals we set at the beginning of this work: performance is improved in all cases and even degrading the memory latency up to 100 cycles, the resulting machine executes all programs faster than the reference machine with less latency. The net consequence of these two facts is that it is possible to lower overall cost by moving from expensive SRAM memories into mass-market low-price DRAM modules, while *improving* performance.

We believe that this two facts should allow for the implementation of vector architectures that spanned a much wider range of price points. That is, using dynamic issue and cheap memory systems and basing a machine on CMOS technology, it should be possible to move vector architectures into the workstation price range while providing much higher bandwidth and performance for vectorizable applications. Also, by using CMOS technology, vector architectures can regain market share in the supercomputer class by providing better performance at lower cost and allowing very large memory systems to be built out of commodity parts.

Finally, we feel that our results should be of use to the growing community of processor architectures implementing some kind of multimedia extensions. As graphics coprocessors and DSP functions are incorporated into general purpose microprocessors, the advantages of vector instruction sets will become more evident. For these types of computations, short vectors will be the norm. In order to sustain high throughput, both from and to special purpose devices such as frame buffers, long memory latencies will have to be tolerated. These types of applications generally require high bandwidths between the chip and the memory system not available in current microprocessors. For both bandwidth and latency problems, dynamically scheduled vector implementations can help achieve improved performance.

9.2 SPILL CODE

A second topic we have been looking at is the negative effect of spill code present in the programs due to the limited number of registers available. We have seen that the 8 vector registers induce a lot of vector spill code. The percentages vary a lot across programs, but 8 out of our 10 benchmarks did have spill code that amounted between 10% and 70% of all memory traffic.

We examined the impact of the vector spill code found in the programs, by performing artificial simulations in which spill code was removed and we evaluated that the negative effect of the vector spill code can go anywhere from a 10% up to a 60%.

To fight the negative effects of this spill code, we have proposed three different techniques that could be add to vector architectures while maintaining code compatibility. The first two were targeted at the DVA and OOOVA and take advantage of the particular hardware found in them. The third technique, a victim cache for registers, is more general and could be applied to any vector machine.

Dynamic Load/Store Elimination in DVA

We have introduced a bypassing technique that allows to copy data from the vector store queue back into the vector load queue. This bypassing is able to service some memory requests without paying the main memory latency and to reduce the total amount of memory traffic and make the processor appear as if, for some cycles, it had two different paths to memory. We have shown how when this bypassing is used in the decoupled architecture, there is an average speedup of 10% over the DVA architecture.

We have also used the natural buffering that the vector store queue provides to eliminate some store traffic. By delaying spill stores as much as possible we favor the likelihood that a subsequent spill to the exact same stack positions renders the first memory operation useless and allows us to discard it. This store elimination technique is able to reduce memory traffic by factors of 1.15 to 1.9.

Dynamic Load Elimination in OOOVA

We have introduced a new technique that uses the renaming mechanism to reduce the amount of load spill traffic. By tagging all our registers with memory information we can detect when a certain load is redundant and its required data is already in some other physical register. Under such conditions, the load can be performed through a simple rename table change. Our simulations have shown that this technique can further improve performance typically by factors of 1.07–1.16 (and as high as 1.78). The dynamic load elimination technique can benefit from more physical registers, since it can cache more data inside the vector register file. Simulations with 32 physical vector registers show that load elimination yields improvements typically in the range 1.10–1.20. Moreover, at 32 registers, load elimination can reduce the total traffic to the memory system by factors ranging between 15–20% and, in some cases, up to 40%.

Dynamic Load/Store Elimination using a victim cache

We have finally considered the usefulness of adding a victim cache next to the register level of a vector processor. We have shown that a victim cache is able to filter this spill code traffic and serve most of the data requested by the processor without accessing main memory.

We have studied several configurations of a write-through victim cache that was able to eliminate most of the load spill traffic. With just 4 slots, simulations have shown that between a 5 and a 20% of load traffic can be eliminated. Although the study was initially targeted at eliminating spill code traffic, when all store traffic is kept in the victim cache, we have seen that some extra traffic elimination is achievable. In particular, hidden spill and reused vectors between iterations of a loop can be captured when all the store traffic is walked through the victim cache. The speedups associated with these traffic reductions go anywhere from 1.05 up to 1.20, with the speedups associated with the all-store model being larger than the spill-only model.

For spill store traffic, we have studied a write-back version of the victim cache. The traffic reductions associated with this write-back victim were much larger than those of the write-through cache, typically in the 10–50% range. The write back cache, though, had two drawbacks. First, by its own nature, it does not allow for precise exceptions if stores are eliminated without updating memory. Second, although the total memory traffic generated was lower than in the write-through case, the corresponding speedups were, in five cases, worse. Our results show that this degradation in speedup is due to invalidations caused by load operations.

Conclusion

The three techniques presented show that it is possible to deal with spill code introduced by limited registers without loosing binary compatibility. This is very important since most of the time running old binaries is a must for new architectures and the performance improvement due to a larger register file, for example, can not be justified if it requires recompilation of old codes.

The three techniques show traffic reductions in the range of 1.1 to 2.0 and can be applied transparently to the user. The speedups associated with these traffic reductions vary: in the DVA, for example, since there is a physical data copy of data being bypassed the speedups are not very large. In the OOOVA machine, the speedups are larger because there is no data copying, only pointer mangling in the rename table, resulting in speedups between 1.2 and 2.0. The victim cache, since it also has data copying, only achieves speedups in the 1.05 to 1.2 range, but has the advantage of being orthogonal to all other proposals: it can be applied to any of the three architectures studied in this thesis or it can be applied to a traditional vector machine.

9.3 FUTURE WORK

Very Wide Machines

A way to fight memory latency not presented in this thesis is to take the "brute force" approach. We saw in section 4.4.1 that interferences between the memory and compute stream produced "holes" in the memory port. One way to reduce this interferences is to add a large number of functional units that will ensure that once data arrives at the processor, there will be no computation stalls that force the memory port to become idle. Preliminary results show that this approach works to a certain extent but it is not able to tolerate as much memory latency as the dynamic scheduling techniques presented.

Machines with memory/compute ratio over 1

It would be interesting to extend this study to machines that have a memory to compute ratio larger than 1, such as Cray's. In these types of machines, the effects of dynamic scheduling can be much different. For example, a Cray machine, having 2 load and 1 store ports should behave differently regarding stalls, since it allows more memory parallelism without using any dynamic scheduling technique.

Simultaneous Vector Multithreading

A straightforward extension to the work presented would be to merge the out-of-order execution and the multithreading technique to yield a Simultaneous Vector Multithreading machine. This combination would allow to completely saturate the memory port while retaining the advantages in latency tolerance, precise exceptions and improved performance of the OOOVA.

On-the-fly multithreading

Also related to multithreading we have the question of how to use a multithreaded machine to speedup a single application instead of improving overall throughput. Extracting threads from sequential applications is currently a topic of active research, but vector semantics can come here to a rescue. If we look at a vector instruction that executes under a vector length of 128, then we could view this instruction as specifying work for 128 different threads. If our machine has two contexts, we could dynamically split this single instruction into two exact instructions each working on 64 elements. Using this technique we could speed up single programs.

Short Vectors

We have seen that the three architectures proposed are able to tolerate a very large memory latency. Therefore it is natural to ask if one could reduce the size of each vector register while still maintaining the same performance. Since decoupling seems to be able to mask many cycles of latency, having smaller registers could translate in two different positive effects: either having the same number of registers at a much lower cost in terms of area or, alternatively, given a certain amount of area we would be able to provide a much larger register file or more queue slots. Studies of both alternatives are currently under way, and show that it is possible to reduce register size down to 32 and even 16 elements while retaining almost the same performance.

Superscalar versus Vector architectures

The performance of highly vectorizable codes on the proposed dynamically scheduled machines opens the question of what architecture paradigm is better to exploit large amounts of ILP. Superscalar architecture have more flexibility but suffer from the Flynn bottleneck: they must fetch and decode large numbers of instructions to use available hardware resources. On the other hand, vector architectures specify large amounts of parallelism in a very concise way.

We are currently studying the relative merits of each type of architectures and relating their behavior to the memory systems they use.

Vector architectures for specific problem domains

The ability of vector machines to use memory bandwidth makes them very attractive for applications that by their own nature require large data movements at high speed: graphics, compression, DSP applications and multimedia can all benefit from vector instruction sets. In fact, most chips that have added multimedia-specific instructions have use a restricted form of vector instructions, but this is likely to change as these chips evolve.

A

A SURVEY OF TRADITIONAL VECTOR ARCHITECTURES

In this appendix we present a brief survey of most register-to-register vector architectures. We have summarized the main characteristics of each machine in a single chart sorted by year of introduction to show the evolution in computing power along the years.

Table A.1 summarizes the main characteristics of the major vector supercomputers build and sold between 1976, the year in which the Cray-1 was introduced, and 1996. Note that we only include register-to-register vector machines. Older memory-to-memory machines, such as the ETA-10 or Cyber-205 are not included.

For each machine in the chart we indicate the year of introduction and the basic cycle time of the machine (in ns units). The next column, labeled "LD/ST paths" indicates the number of ports connecting the memory system to the vector register files. There are three kinds of ports: an "L" by itself indicates a stand-alone load port. An "S" by itself indicates a stand-alone store port, while an "LS" indicates a bidirectional port that can do both loads and stores. For example, the SX-2 had 2 different ports: one port was a load-only port while

Machine	Year intro	Cycle time	LD/ST paths	Functional Units	Pipes	Flops/ cycle	Words/ cycle
Cray-1	1976	12.5	LS	A,M,R,I,L,S	1	2	1
Fujitsu VP200	1982	7.0	LS,LS	A/L,M,D	1	3	2
Cray X-MP	1983	9.5	L,L,S	A,M,R,I,L,L,S,P	1	2	2+1
Hitachi S810/20	1983	14.0	L,L,L,LS	A/L,A/L,M/D+A,M+A	1	4	4 or 1
NEC SX-2	1984	6.0	L,LS	A,M/D,L,S	4	16	8 or 4
Cray-2	1985	4.1	LS	A,M/R/Q,I,L	1	2	1
Hitachi S820/80	1987	4.0	L,LS	A/L,M+A,D	4	12	8 or 4
Cray Y-MP	1988	6.3	L,L,S	A,M/L,R,I,L,S,P	1	2	2+1
Fujitsu VP2600	1989	3.2	LS,LS	M+A/L,M+A/L,D	4	16	8
NEC SX-3	1990	2.9	L,L,S	A/S,A/S,M/L,M/L	4	16	8+4
Cray C-90	1992	4.0	L,L,S	A,M/L,R,I,L,S,P	2	4	4+2
NEC SX-4	1996	8.0	LS,LS	A/S,M,D,L	8	16	16

Table A.1 Functional unit characteristics of several vector supercomputers.

the other was a bidirectional load/store port. Similarly, the Cray C90 has three ports: two load-only ports and one store-only port.

Under the column "Functional Units" we indicate all the functional units present in the architecture. We have used the following keys to indicate the several types of arithmetic units:

A = Floating point add
D = Floating point divide
I = Integer add
L = Logical
M = Floating point multiply
P = Population Count/Parity
R = Reciprocal Approximation
Q = Reciprocal Square Root
S = Shift

Independent units shown in table A.1 are separated by commas; for example, the Cray-1 had 6 independent units (not all where floating point, though). The S810/20 has four independent units. Since each functional unit can provide more than one operation, we have indicated this feature using a slash ("/") mark. For example, the SX-4 has a unit ("A/S") that can perform *either* floating point additions or shift operations. Some machines offer cascaded units. That is, units that are not directly connected to the register file but that hang off another functional unit. This is the case for the Hitachi S810/20: it has an add unit cascaded after a multiply unit ("M+A") and also has another add unit cascaded after a multiply/divide unit ("M/D+A").

Column "Pipes" indicates the number of replicated pipes existing in the maximum configuration of the machine. For example, the Fujitsu VP2600 is a 4-pipe machine, indicating that each of its main vector resources was replicated four times. Using this column it is easy to see that the VP2600 can transfer 8 words from registers to memory and vice-versa every cycle (two load/store ports replicated four times).

Finally, columns "Flops/Cycle" and "Words/Cycle" indicate the peak number of floating point operations that the machine could do per cycle and the maximum data transfer rate (also per cycle) between the register file and main memory. For machines having asymmetric load/store ports, the following notation has been used. For the SX-2, since it has 1 load-only port and one load/store port, its maximum transfer rate is 8 words if doing loads and 4 words if doing stores. We have noted this situation using the notation "8 or 4". For machines having independent load/store ports, such as the Y-MP, the notation "2+1" indicates that it can do 2 loads and 1 store per cycle.

LIST OF FIGURES

Chapter 5

LIST OF TABLES

Chapter 8

Chapter 9

Appendix A

REFERENCES

[AAD+93] Tom Asprey, Gregory S. Averill, Eric DeLano, Russ Mason, Bill Weiner, and Jeff Yetter. Performance Features of the PA7100 Microprocessor. *IEEE Micro*, pages 22–35, June 1993.

[ABI+95] Krste Asanovic, James Beck, Bertrand Irissou, Brian Kingsbury, Nelson Morgan, and John Wawrzynek. The T0 Vector Microprocessor. In *Hot Chips VII*, pages 187–196, August 1995.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conference Proceedings, 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, June 11–15, 1990. ACM SIGARCH. *Computer Architecture News,* 18(3), September 1990.

[Aga92] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[ALKK90] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 18(2), June 1990.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings AFIPS 1967 Spring Joint Computer Conference*, pages 483–485, April 1967.

[AP93] Anant Agarwal and Steven D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 179–190, San Diego, California, May 17–19, 1993. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 21(2), May 1993.

[Arn82] C. N. Arnold. Performance evaluation of three automatic vectorizer packages. *ICPP82*, pages 235–242, 1982.

[Arn83] C. Arnold. Vector Optimization on the Cyber 205. In *International Conference on Parallel Processing*, August 1983.

[AST67] D.W. Anderson, F. J. Sparacio, and F. M. Tomasulo. The IBM System/360 model 91: Machine philosophy and instruction handling. *IBM Journal of Research and Development*, 11:8–24, January 1967.

[BBB+95] Bradley J. Benschneider, Andrew J. Black, William J. Bowhill, Sharon M. Britton, Daniel E. Dever, Dale R. Donchin, Robert J. Dupack, Richard M. Fromm, Mary K. Gowan, Paul E. Gronowski, Michael Kantrowitz, Marc E. Lamere, Shekhar Metha, Jeanne E. Meyer, Robert O. Mueller, Andy Olesin, Ronald P. Preston, Donald A. Priore, Sribalan Santhanam, Michael J. Smith, and Gilber M. Wolrich. A 300-MHz 64-b Quad-Issue CMOS RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 30(11):1203–1214, November 1995.

[BCcRJ+94] David F. Bacon, Jyh-Herng Chow, Dz ching R. Ju, Kalyan Muthukumar, and Vivek Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94 – Integrated Solutions*, pages 270–282, Toronto, Ontario, October 31–November 3, 1994. IBM Centre for Advanced Studies.

[BCG+91] D. Bradley, G. Cybenko, G. Gao, J. Larson, F. Ahmad, J. Golab, and M. Straka. Supercomputer Workload Decomposition and Analysis. In *Proceedings of the International Conference on Supercomputing*, pages 458–467, 1991.

[BCL91] M. Berry, G. Cybenko, and J. Larson. Scientific benchmarks characterization. *Parallel Computing*, 17(10 & 11):1173–1194, December 1991.

[BDHS94] Jama Barreh, Sudhir Dhawan, Troy Hicks, and David Shippy. The POWER2 Processor. In *COMPCON*, 1994.

[Ber89] M. Berry. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, pages 5–40, Fall 1989.

[BGK96] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society TCCA.

[BKB90] Håkon O. Bugge, Ernst H. Kristiansen, and Bjorn O. Bakka. Trace-driven simulations for a two-level cache design in open bus systems. In *Proceedings of* *the 17th Annual International Symposium on Computer Architecture*, pages 250–259, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 18(2), June 1990.

[BL81] D. K. Bradley and J. L. Larson. A parallelism-based analytical approach to performance evaluation using application programs. *Proceedings of the IEEE*, 8:1126–1135, August 1981.

[BR92] B. Boothe and A. Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 214–223, May 1992.

[BRT93] Peter L. Bird, Alasdair Rawsthorne, and Nigel P. Topham. The effectivness of decoupling. In *Proceedings of the Seventh International Conference on Supercomputing*, Tokyo, Japan, July 1993. IEEE Computer Society Press.

[BW83] W. C. Brantley and Joseph Weiss. Organization and architecture tradeoffs in FOM. In *IEEE International Workshop on Computer Systems Organization*, March 1983.

[BW88] Jean-Loup Baer and Wen-Hann Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 16(2), May 1988.

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Boston, Massachusetts, October 12–15, 1992. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 20, October 1992; *Operating Systems Review,* 26, October 1992; *SIGPLAN Notices,* 27(9), September 1992.

[CB94a] Tien-Fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching strategies. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 223–232, 1994.

[CB94b] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of ALPHA AXP performance using TP and SPEC workloads. *International Symposium on Computer Architecture*, pages 60–70, 1994.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. *Pro-*

ceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.

[CDL88] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Proceedings of Supercomputing'88*, pages 98–105, Orlando, Florida, November 1988. IEEE Computer Society Press.

[CDN92] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, Portland, Oregon, December 1–4, 1992. IEEE Computer Society TC-MICRO and ACM SIGMICRO. SIG MICRO Newsletter 23(1–2), December 1992.

[CG95] Brad Calder and Dirk Grunwald. Next cache line and set prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 287–296, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 23(2), May 1994.

[CGMW88] Mike Chastain, Gary Gostin, Jim Mankovich, and Steve Wallach. The CONVEX C240 architecture. In *Proceedings of Supercomputing'88*, pages 321–329, Orlando, Florida, November 1988. IEEE Computer Society Press.

[CHM89] L. D. Coraor, P. T. Hulina, and D. N. Mannai. A queue based instruction cache memory. In *Proceedings International Symposium on Computer Architecture and Digital Signal Processing*, pages 28–36, October 1989.

[CKDK91] R.F. Cmelik, S.I. Kong, D.R. Ditzel, and E.J. Kelly. An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, Santa Clara, CA, April 1991.

[CM95] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, California, June 18–21, 1995. *SIGPLAN Notices,* 30(6), June 1995.

[CNO+87] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David P. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, Palo Alto, California, October 5–8, 1987. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 15(5), October 1987; *Operating Systems Review,* 21(4), October 1987; *SIGPLAN Notices,* 22(10), October 1987.

[Con92] Convex Press, Richardson, Texas, U.S.A. *CONVEX Architecture Reference Manual (C Series)*, sixth edition, April 1992.

[Cra90] Cray Research, Inc. *Cray Y-MP System Architecture for Systems Analysts*, TR-YSASA Revision E edition, April 1990.

[CS81] E. U. Cohler and J. E. Storer. Functionally parallel architectures for array processors. *Computer*, 14:28–36, September 1981.

[CS95] R. Colwell and R. Steck. A 0.6-μm BiCMOS Microprocessor with Dynamic Execution. In *Proceedings of the International Solid-State Circuits Conference*, pages 176–177, Piscataway, N.J., 1995. IEEE.

[DA92] K. Diefendorff and M. Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(4):40–63, April 1992.

[DC88] Henry Dietz and Chi-Hung Chi. CRegs: A new kind of memory for referencing arrays and pointers. In *Proceedings of Supercomputing'88*, pages 360–367, Orlando, Florida, November 1988. IEEE Computer Society Press.

[Del91] G. Delic. Performance analysis of a 24 code sample on Cray X/Y-MP at the Ohio Supercomputer Center. In *Proceedings of the 5th SIAM Conference on Parallel Processing for Scientific Applications*, pages 530–535, 1991.

[DNS95] Trung A. Diep, Christopher Nelson, and John Paul Shen. Performance evaluation of the PowerPC 620 microarchitecture. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 163–174, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 23(2), May 1994.

[DO95] Mikhail N. Dorojevets and Vojin G. Oklobdzija. Multithreaded Decoupled Architecture. *International Journal of High Speed Computing*, 7(3):465–480, 1995.

[Don93] J. J. Dongarra. Performance of various computers using standard linear equations software in a fortran environment. Technical Report CS-89-85, Univeristy of Tennesse, 1993.

[EJK+96] Richard J. Eickemeyer, Ross E. Johnson, Steve R. Kunkel, Mark S. Squillante, and Shiafun Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 203–212. ACM Press, May 1996.

[EJL88] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Squeezing more CPU performance out of a Cray-2 by vector block scheduling. In *Proceedings of Supercomputing'88*, pages 237–246, Orlando, Florida, November 1988. IEEE Computer Society Press.

[EM94] Roger Espasa and Xavier Martorell. Dixie: a trace generation system for the C3480. Technical Report CEPBA-RR-94-08, Universitat Politècnica de Catalunya, 1994.

[EML88] C. Eoyang, R. H. Mendez, and O. M. Lubeck. The Birth of the Second Generation: The Hitachi S-820/80. In *Proceedings of Supercomputing'88*, pages 296–303, Orlando, Florida, November 1988. IEEE Computer Society Press.

[EP88] Richard J. Eickenmeyer and Janak H. Patel. Performance evaluation of on-chip register and cache organizations. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 64–72, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 16(2), May 1988.

[Esp95] Roger Espasa. JINKS: A parametrizable simulator for vector architectures. Technical Report UPC-CEPBA-1995-31, Universitat Politècnica de Catalunya, 1995.

[EV95] Roger Espasa and Mateo Valero. Instruction level characterization of the Perfect Club programs on a vector computer. In *XV International Conference of the Chilean Computation Society*, pages 198–209. SCCC Press, January 1995.

[EV96] Roger Espasa and Mateo Valero. Decoupled vector architectures. In *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, pages 281–290. IEEE Computer Society Press, Feb 1996.

[EVP+95] R. Espasa, M. Valero, D. Padua, M. Jiménez, and E. Ayguadé. Quantitative analysis of vector code. In *Euromicro Workshop on Parallel and Distributed Processing.* IEEE Computer Society Press, January 1995.

[Fat89] R. A. Fatoohi. Vector performance analysis of three supercomputers: Cray 2, cray Y-MP, and ETA 10-Q. In *Proceedings of the Supercomputing 89*, pages 779–788, Reno, NV USA, 1989. ACM Press , New York, NY , USA.

[Fat90] Rod A. Fatoohi. Vector performance analysis of the NEC SX-2. In *Conference Proceedings, 1990 International Conference on Supercomputing*, pages 389–400, Amsterdam, The Netherlands, June 11–15, 1990. ACM SIGARCH. *Computer Architecture News,* 18(3), September 1990.

[FD95] Stuart Fiske and William J. Dally. Thread prioritization: A thread scheduling mechanism for multiple-context parallel processors. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 210–221, Raleigh, North Carolina, January 22–25, 1995. IEEE Computer Society TCCA.

[FP91a] Mathew K. Farrens and Andrew R. Pleszkun. Implementation of the PIPE processor. *Computer*, pages 65–70, January 1991.

[FP91b] John W. C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. *International Symposium on Computer Architecture*, pages 54–63, 1991.

[GAG94] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 85–94, San Jose, California, November 30–December 2, 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[GB96] Manu Gulati and Nader Bagherezadeh. Performance study of a multithreaded superscalar microprocessor. In *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, pages 291–301. IEEE Computer Society Press, Feb 1996.

[GBH96] Ralf Gruber, James Brunson, and Michael Hodous. Preliminary Benchmark Measurments on the NEC SX-4. *SPEEDUP Journal*, 9(2):76–79, 1996.

[GHG+91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, Toronto, Ontario, May 27–30, 1991. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 19(3), May 1991.

[GHL+85] J.R. Goodman, J.T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young. PIPE: A VLSI Decoupled Architecture. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 20–27, June 1985.

[GHW90] Guang R. Gao, Herbert H. J. Hum, and Yue-Bong Wong. Towards efficient fine-grain software pipelining. In *Conference Proceedings, 1990 International Conference on Supercomputing*, pages 369–379, Amsterdam, The Netherlands, June 11–15, 1990. ACM SIGARCH. *Computer Architecture News,* 18(3), September 1990.

[GNL95] R. Govindarajan, S. S. Nemawarkar, and Philip LeNir. Design and performance evaluation of a multithreaded architecture. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 298–307, Raleigh, North Carolina, January 22–25, 1995. IEEE Computer Society TCCA.

[Goo83] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, Stockholm, Sweden, June 13–17, 1983. *Computer Architecture News,* 11(3), June 1983.

[GS92] J.D. Gee and A. J. Smith. The performance impact of vector processor caches. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume 1, pages 437–449, January 1992.

[Han96] Craig Hansen. MicroUnity's MediaProcessor Architecture. *IEEE Micro*, pages 34–41, August 1996.

[HC91] P. T. Hulina and L. D. Coraor. Performance Analysis of and address generation coprocessor. In *International Conference on Parallel Processing, vol I*, pages 136–143, August 1991.

[Hil87] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California at Berkeley Computer Science Division, 1987. Tech. Rep. UCB/CSD 87/381 (November).

[Hil88] Mark D. Hill. A Case for Direct Mapped Caches. *IEEE Computer*, 21(12):25–40, December 1988.

[HKN+92] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 136–145, 1992.

[HMD+86] Christopher Huston, Thomas Macke, James Davies, Michael Wolfe, and Bruce Leasure. The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer. In *Proceedings of the International Conference on Parallel Processing*, pages 827–832, August 1986.

[HMHY93] T. Hashimoto, K. Murakami, T. Hironaka, and H. Yasuura. A microvectorprocessor architecture – performance modeling and benchmarking –. In *Proceedings of the International Conference on Supercomputing*, pages 308–317, 1993.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[Hsu94] P.Y.T Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[HT72] R. G. Hintz and D. P. Tate. Control data STAR-100 processor design. In *Proc. Compcon 72*, pages 1–4, New York, 1972. IEEE Computer Society Conf. 1972, IEEE.

[Hun95] Doug Hunt. Advanced Performance Features of the 64-bit PA-8000. In *COMPCON*, 1995.

[IMS88] M. Ishiguro, M. Makino, and N. Shinozawa. Vector and Parallel Processing of the Nuclear Reactor Transient Analysis Code RELAPS. In *Proceedings of Supercomputing'88*, pages 230–236, Orlando, Florida, November 1988. IEEE Computer Society Press.

[IW91] Akihiro Iwaya and Tadashi Watanabe. The parallel processing feature of the NEC SX-3 supercomputer system. *Intl. Journal of High Speed Computing*, 3(3&4):187–197, 1991.

[Jai91] Suneel Jain. Circular scheduling: A new technique to perform software pipelining. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 219–228, Toronto, Ontario, June 26–28, 1991. *SIGPLAN Notices*, 26(6), June 1991.

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

[Jou89] Norman P. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, 38(12):1645–1658, 1989.

[Jou90] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990.

[Jou93] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201, San Diego, California, May 17–19, 1993. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News*, 21(2), May 1993.

[JW89] Norman P. Jouppi and David W. Wall. Available instruction level parallelism for superscalar and superpipelined machines. *ASPLOS*, pages 272–282, 1989.

[KD92] Stephen W. Keckler and William J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the*

19th Annual International Symposium on Computer Architecture, pages 202–213, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 20(2), May 1992.

[Kel96] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. In *Microprocessor Forum*, October 1996.

[KELS62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level Storage system. *IRE Transactions on Electronic Computers*, EC-11:223–235, April 1962. Also appears in [SBN82, pages 135–148].

[KG92] Tokuzo Kiyohara and John C. Gyllenhaal. Code scheduling for VLIW/superscalar processors with limited register files. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 197–201, Portland, Oregon, December 1–4, 1992. IEEE Computer Society TC-MICRO and ACM SIGMICRO. SIG MICRO Newsletter 23(1–2), December 1992.

[KHC92] Lizy Kurian, Paul T. Hulina, and Lee D. Coraor. Memory Latency Effects in Decoupled Architectures with a single data memory module. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 237–245, Gold Coast, Australia, May 1992.

[KHC94] Lizy Kurian, Paul T. Hulina, and Lee D. Coraor. Memory Latency Effects in Decoupled Architectures. *IEEE Transactions on Computers*, 43(10):1129–1139, October 1994.

[KHG92] S. Kawabe, M. Hirai, and S. Goto. Hitachi S-820 Supercomputer System. In Raul Mendez, editor, *High Performance Computing – Research and Practice in Japan*, Wiley Professional Computing, pages 35–53. John Wiley & Sons Ltd., 1992.

[KIS+94] Katsuyoshi Kitai, Tadaaki Isobe, Tadayuki Sakakibara, Shigeko Yazawa, Yoshiko Tamaki, Teruo, and Kouichi Ishii. Distributed storage control unit for the Hitachi S-3800 multivector supercomputer. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, July 1994.

[KM92] Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the International Conference on Supercomputing*, pages 323–334, July 1992.

[KSF+94] L.I. Kontothanassis, R. A. Sugumar, G. J. Faanes, J. E. Smith, and M. L. Scott. Cache performance in vector supercomputers. In *Proceedings of Supercomputing'94*, Washington D.C., November 1994. IEEE Computer Society Press.

[Lam88] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, June 1988.

[LB96] Beng-Hong Lim and Ricardo Bianchini. Limits on the Performance Benefits of MUltithreading and Prefetching. In *Proceedings of the SIGMETRICS Conference*, pages 37–46, PA, U.S.A, June 1996.

[LC95] Yamin Li and Wanming Chu. The effects of STEF in finely parallel multithreaded processors. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 318–325, Raleigh, North Carolina, January 22–25, 1995. IEEE Computer Society TCCA.

[Lee92] Corinna G. Lee. *Code Optimizers and Register Organizations for Vector Architectures*. PhD thesis, University of California at Berkeley, 1992.

[Lee96] Ruby B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, pages 51–59, August 1996.

[Li95] Wei Li. Compiler cache optimizations for banded matrix problems. In *Conference Proceedings, 1995 International Conference on Supercomputing*, pages 21–30, Barcelona, Spain, July 3–7, 1995. ACM SIGARCH.

[LJ90] Carl E. Love and Harry F. Jordan. An investigation of static versus dynamic scheduling. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 192–201, May 1990.

[LRW91a] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[LRW91b] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review,* 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[LS84] J.K.F. Lee and A.J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.

[MBG95] S.C. McMahan, M. Bluhm, and R.A. Garibay. 6x86: The Cyrix Solution to Executing x86 Binaries on a High Performance Microprocessor. *Proceedings of the IEEE*, 83(12):1664–1672, December 1995.

[MCmWH+92] Scott A. Mahlke, William Y. Chen, Wen mei W. Hwu, B. Ramakrisna Rau, and Michael S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247, Boston, Massachusetts, October 12–15, 1992. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 20, October 1992; *Operating Systems Review,* 26, October 1992; *SIGPLAN Notices,* 27(9), September 1992.

[ML93] Larry McMahan and Ruby Lee. Pathlengths of SPEC benchmarks for PA-RISC, MIPS, and SPARC. *COMPCON*, 1993.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[MLR90] Malony, Larson, and Reed. Tracing Application Program Execution on the Cray X-MP and Cray 2. In *Proceedings of Supercomputing'90.* IEEE Computer Society Press, November 1990.

[MNT91] Kenichi Miura, Hiroshi Nagakura, and Hiroshi Tamura. VP2000 Series Dual Scalar and Quadruple Scalar Models Supercomputer Systems– A New Concept in Vector Processing –. In *compcon91*, pages 294–302, 1991.

[MSAD90] William Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. The effects of memory latency and fine-grain parallelism on astronautics ZS-1 performance. In *Twenty-Third Annual Hawaii International Conference on System Sciences*, January 1990.

[MSAD91a] W. Mangione-Smith, S.G. Abraham, and E.S. Davidson. Vector register design for polycyclic vector scheduling. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 154–163, Santa Clara, CA, April 1991.

[MSAD91b] William Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. A Performance Comparison of the IBM RS/6000 and the Astronautics ZS-1. *Computer*, January 1991.

[NEC95] NEC Corporation. *NEC SX-4 SERIES – General Description Manual*, Edition GUZ41E-1 edition, June 1995.

[NKT+95] Yasuhiko Nakashima, Toshiaki Kitamura, Hideo Tamura, Masaaki Takiuchi, and Kenichi Miura. Scalar processor of the VPP500 parallel supercomputer.

In *Conference Proceedings, 1995 International Conference on Supercomputing*, pages 348–356, Barcelona, Spain, July 3–7, 1995. ACM SIGARCH.

[NMP+96] L. T. Nguyen, M. Mohamed, H. Park, Y. Pai, R. Wong, A. Qureshi, P. Psong, F. Valesco, H. D. Truong, and C. Reader. Multi-media Signal Processor (MSP) summary. In *Hot Chips VIII*, pages 217–226, Palo Alto, CA, August 1996. Samsung Semiconductor.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 20(2), May 1992.

[NPR88] I. M. Navon, P. K. H. Phua, and M. Ramamurthy. Vectorization of Conjugate Gradient-Methods for Large-Scale Minimization. In *Proceedings of Supercomputing'88*, pages 410–418, Orlando, Florida, November 1988. IEEE Computer Society Press.

[Oed92] W. Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18(8):947–954, August 1992.

[OW93] Wilfried Oed and Martin Walker. An overview of Cray Research computers including the Y-MP/C90 and the new MPP T3D. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 271–272, Velen, Germany, June 30–July 2, 1993. SIGACT and SIGARCH.

[Pap96] David B. Papworth. Tuning the Pentium Pro Microarchitecture. *IEEE Micro*, pages 8–15, April 1996.

[PAY96] Dave Patterson, Tom Anderson, and Kathy Yelick. A Case for Intelligent DRAM: IRAM. In *Hot Chips VIII*, August 1996.

[PB88] D. V. Pryor and P. J. Burns. Vectorized Monte Carlo Molecular Aerodynamics Simulation of the Reyleigh Problem. In *Proceedings of Supercomputing'88*, pages 384–391, Orlando, Florida, November 1988. IEEE Computer Society Press.

[PGH+87] A. R. Pleszkun, J. R. Goodman, W-C. Hsu, R. T. Joersz, G. Bier, P. Woest, and P. B. Schechter. WISQ: A Restartable Architecture Using Queues. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 290–299, June 1987.

[PHH88] Steven Przybylski, Mark Horowitz, and John Hennessy. Performance tradeoffs in cache design. In *Proceedings of the 15th Annual International Symposium* *on Computer Architecture*, pages 290–298, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 16(2), May 1988.

[PHH89] Steven Przybylski, Mark Horowitz, and John Hennessy. Characteristics of performance-optimal multi-level cache hierarchies. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 114–121, Jerusalem, Israel, May 28–June 1, 1989. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 17(3), June 1989.

[PK94] Subbarao Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 22(2), April 1994.

[PM86] Robert Perron and Craig Mundie. The architecture of the Alliant FX/8 computer. In *Digest of Papers: Compcon 86*, pages 390–394, Washington, D.C., March 1986.

[Pri96] Betty Prince. *High Performance Memories.* Wiley & Sons, Ltd., 1996.

[PW96] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, pages 42–50, August 1996.

[Ram94] J. Ramanujam. Optimal software pipelining of nested loops. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 335–342, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, San Jose, California, November 30–December 2, 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[RLTS92] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 283–299, San Francisco, California, June 17–19, 1992. *SIGPLAN Notices,* 27(7), July 1992.

[RR94] K.A. Robbins and S Robbins. Relationship between average and real memory behavior. *The Journal of Supercomputing*, 8(3):209–232, November 1994.

[Rus78] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[SBN82] D. P. Siewiorek, C. G. Bell, and A. Newell. *Computer Structures: Principles and Examples.* Mc-Graw Hill, New York, 1982.

[SD84] G. S. Sohi and E. S. Davidson. Performance of the Structured Memory Access (SMA) Architecture. In *Proceedings of the International Conference on Parallel Processing*, pages 506–513, August 1984.

[SDV+87] James E. Smith, G.E. Dermer, B.D. Vanderwarn, S.D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon. The ZS-1 Central Processor. In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–204. CS press, 1987.

[SEI+95] M. Simone, A. Essen, A. Ike, A. Krishnamoorthy, T. Maruyama, N. Patkar, M. Ramaswami, M. Shebanow, V. Thirumalaiswamy, and D. Tovey. Implementation trade-offs in using a restricted data flow architecture in a high performance RISC microprocessor. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 151–162, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 23(2), May 1994.

[Sez93] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 169–178, San Diego, California, May 17–19, 1993. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 21(2), May 1993.

[SG83] James E. Smith and James R. Goodman. A study of instruction cache organizations and replacement policies. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 132–137, Stockholm, Sweden, June 13–17, 1983. *Computer Architecture News,* 11(3), June 1983.

[SH91] Willi Schönauer and Hartmut Häfner. Supercomputers: Where are the lost cycles ? *Supercomputing*, 1991.

[SH94] Willi Schönauer and Hartmut Häfner. Explaining the gap between theoretical peak performance and real performance for supercomputer architectures. *Scientific Programming*, 3:157–168, 1994.

[SHH90] J. E. Smith, W. C. Hsu, and C. Hsiung. Future general purpose supercomputer architectures. In *Proceedings of Supercomputing'90.* IEEE Computer Society Press, November 1990.

[Shi82] R. R. Shively. Architecture of a programmable digital signal processor. *IEEE Transactions on Computers*, C-31, January 1982.

[SJH89] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on multiple instruction issue. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, Boston, Massachusetts, 1989.

[Smi82a] A.J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Smi82b] James E. Smith. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, 1982.

[Smi84] James E. Smith. Decoupled Access/Execute Computer Architectures. *ACM Transactions on Computer Systems*, 2:289–308, November 1984.

[Smi89] James E. Smith. Dynamic Instruction Scheduling and the Astronautics ZS-1. *Computer*, 22(7), July 1989.

[SPKG83] J. E. Smith, A. R. Pleszkun, R. H. Katz, and J. R. Goodman. PIPE: A high-performance VLSI architecture. In *IEEE International Workshop on Computer System Organization*, March 1983.

[SS95] James E. Smith and Gurinadar S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, 83(12), December 1995.

[SW88] M. L. Simmons and H. J. Wasserman. Performance Comparison of the Cray-2 and Cray X-MP/416 Supercomputers. In *Proceedings of Supercomputing'88*, pages 288–295, Orlando, Florida, November 1988. IEEE Computer Society Press.

[SWL+92] Margaret L. Simmons, Harvey J. Wasserman, Olaf M. Lubeck, Christopher Eoyang, Raul Mendez, Hiroo Harada, and Misako Ishiguro. A performance comparison of four supercomputers. *Communications of the ACM*, 35(3):116–124, 1992.

[SWP86] James E. Smith, Shlomo Weiss, and Nicholas Y. Pang. A Simulation Study of Decoupled Architecture Computers. *IEEE Transactions on Computers*, C-35(8):692–702, August 1986.

[SZTH92] Bogong Su, Wei Zhao, Zhizhong Tang, and Stanley Habib. A VLIW architecture for optimal execution of branch-intensive loops. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 119–124, Portland, Oregon, December 1–4, 1992. IEEE Computer Society TC-MICRO and ACM SIGMICRO. SIG MICRO Newsletter 23(1–2), December 1992.

[Tan96] Philipp Tannenbaum. HNSX Supercomputers Inc.; Marketing Group Director, 1996. Private Communication.

[TDT88] Juho Tang, Edward S. Davidson, and Johau Tong. Polycyclic vector scheduling vs. chaining on 1-port vector supercomputers. In *Proceedings of Supercomputing'88*, pages 122–129, Orlando, Florida, November 1988. IEEE Computer Society Press.

[TE94] Radhika Thekkath and Susan J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 176–186, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 22(2), April 1994.

[TEE+95] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[TEL96] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 191–202. ACM Press, May 1996.

[TGN95] M. Tremblay, D. Greenley, and K. Normoyle. The design of the microarchitecture of UltraSPARC-I. *Proceeddings of the IEEE*, 83(12):1653–1663, December 1995.

[TM95] Nigel Topham and Kenneth McDougall. Performance of the Decoupled ACRI-1 Architecture: the Perfect Club. In *High Performance Computing and Networking-Europe*, pages 472–480, May 1995. LNCS #919, Springer-Verlag.

[Tom67] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[TONH96] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, pages 10–20, August 1996.

[TOU92] M. Takahashi, Y. Oinaga, and K. Uchida. Fujitsu VP2000 Series. In Raul Mendez, editor, *High Performance Computing – Research and Practice in Japan*, Wiley Professional Computing, pages 7–20. John Wiley & Sons Ltd., 1992.

[Tzi91] Tzi-cker Chiueh. Multithreaded vectorization. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 352–361, 1991.

[Uch91] Nobuo Uchida. FUJITSU VP2000 series supercomputers. *International Journal of High Speed Computing*, 3(3 & 4):169–185, 1991.

[UIT94] T. Utsumi, M. Ikeda, and M. Takamura. Architecture of the VP500 Parallel Supercomputer. In *Proceedings of Supercomputing'94*, pages 478–487, Washington D.C., November 1994. IEEE Computer Society Press.

[Vaj91] Sriram Vajapeyam. *Instruction-Level Characterization of the Cray Y-MP processor*. PhD thesis, University of Wisconsin, Madison, 1991.

[VH92] Sriram Vajapeyam and Wei-Chung Hsu. On the instruction-level characteristics of scalar code in highly-vectorized scientific applications. *IEEE Micro 25*, pages 20–28, 1992.

[VSH91] Sriram Vajapeyam, Gurindar S. Sohi, and Wei-Chung Hsu. An empirical study of the Cray Y-MP processor using the PERFECT Club benchmarks. *International Symposium on Computer Architecture*, pages 170–179, 1991.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Santa Clara, CA, April 1991.

[Wat72] W. Watson. The TI-ASC, A highly modular and flexible super computer architecture. *Proc. AFIPS*, 41, pt. 1:221–228, 1972.

[Wat91] Tadashi Watanabe. The NEC SX-3 Supercomputer System. In *compcon91*, pages 303–308, 1991.

[WBL89] Wen-Hann Wang, Jean-Loup Baer, and Henry M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 140–148, Jerusalem, Israel, May 28–June 1, 1989. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 17(3), June 1989.

[WI92] T. Watanabe and A. Iwaya. The NEC SX-3 Supercomputer Series. In Raul Mendez, editor, *High Performance Computing – Research and Practice in Japan*, Wiley Professional Computing, pages 21–33. John Wiley & Sons Ltd., 1992.

[Wil65] Maurice Wilkes. Slave Memories and Dynamic Storage Allocation. *IRE Transactions on Electronic Computers*, EC-14(2):270–271, April 1965.

[WJ91] D. Windheiser and W. Jalby. Behavioral Characterization of Decoupled Access/Execute Architectures. In *Proceedings of the Fifth ACM International Conference on Supercomputing*, Cologne, Germany, June 1991. IEEE Computer Society Press.

[WKI86] T. Watanabe, H. Katayama, and A. Iwaya. Introduction of the NEC Supercomputer SX System. In Sidney Fernbach, editor, *Supercomputers – Class VI Systems, Hardware and Software*, pages 153–167. North-Holland, Amsterdam, Netherlands, 1986.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[WMK90] Williams, Myers, and Koskela. The Characterization of Two Scientific Workloads Using the Cray X-MP Performance Monitor. In *Proceedings of Supercomputing'90.* IEEE Computer Society Press, November 1990.

[WPS95] Ted Williams, Niteen Patkar, and Gene Shen. SPARC64: A 64-b 64-Active-Instruction Out-of-Order-Execution MCM Processor. *IEEE Journal of Solid-State Circuits*, 30(11), November 1995.

[WS91] Andrew Wolfe and John P. Shen. A variable instruction stream extension to the VLIW architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review,* 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[WSY95] Hong Wang, Tong Sun, and Qing Yang. CAT – caching address tags: A technique for reducing area cost of on-chip caches. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 381–390, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 23(2), May 1994.

[Wul92] W. A. Wulf. Evaluation of the WM Architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 382–390, Gold Coast, Australia, May 1992.

[Yag96] Keneth C. Yager. The Mips R10000 Superscalar Microprocessor. *IEEE Micro*, pages 28–40, April 1996.

[Yan93] Qing Yang. Introducing a new cache design into vector computers. *IEEE Transactions on Computers*, 52(12):1411–1424, December 1993.

[YG84] H. C. Young and J. R. Goodman. A Simulation Study of Architectural Data Queues and Prepare-to-Branch Instruction. In *International Conference on*

Computer Design: VLSI in Computers, pages 544–549, Port Chester, NY, 1984.

[YN95] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance thorugh multistreaming. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, 1995.

[You85] H. C. Young. *Evaluation of a Decoupled Computer Architecture and the Design of a Vector Extension.* PhD thesis, University of Wisconsin ar Madison, July 1985.

[You88] H. C. Young. Code Scheduling Methods for Some Architectural Features in PIPE. *Microprocessing and Microprogramming*, 22:39–63, 1988.

[YZ91] L. C. Young and S. E. Zarantonello. High-Performance Vector Processing in Reservoir Simulation. In *Proceedings of Supercomputing'91*, pages 304–315, Albuquerque, New Mexico, November 1991. IEEE Computer Society Press.