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

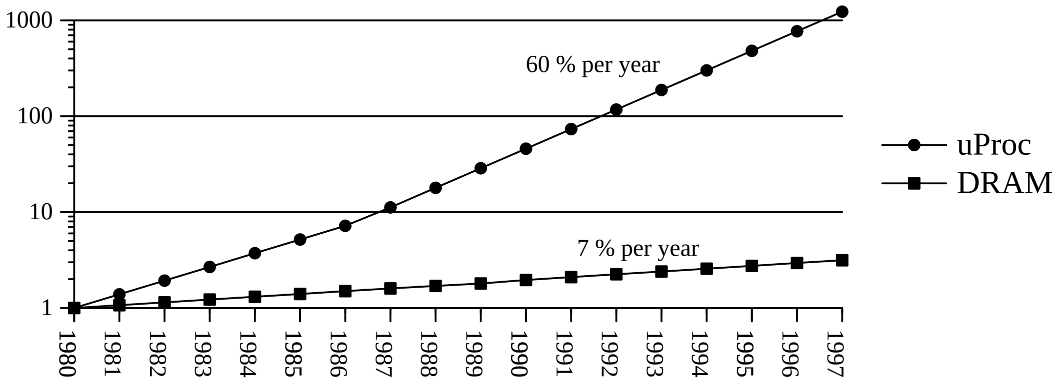

Figure 1.1 Relative evolution of microprocessor and DRAM speeds (source [PAY96]).

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].

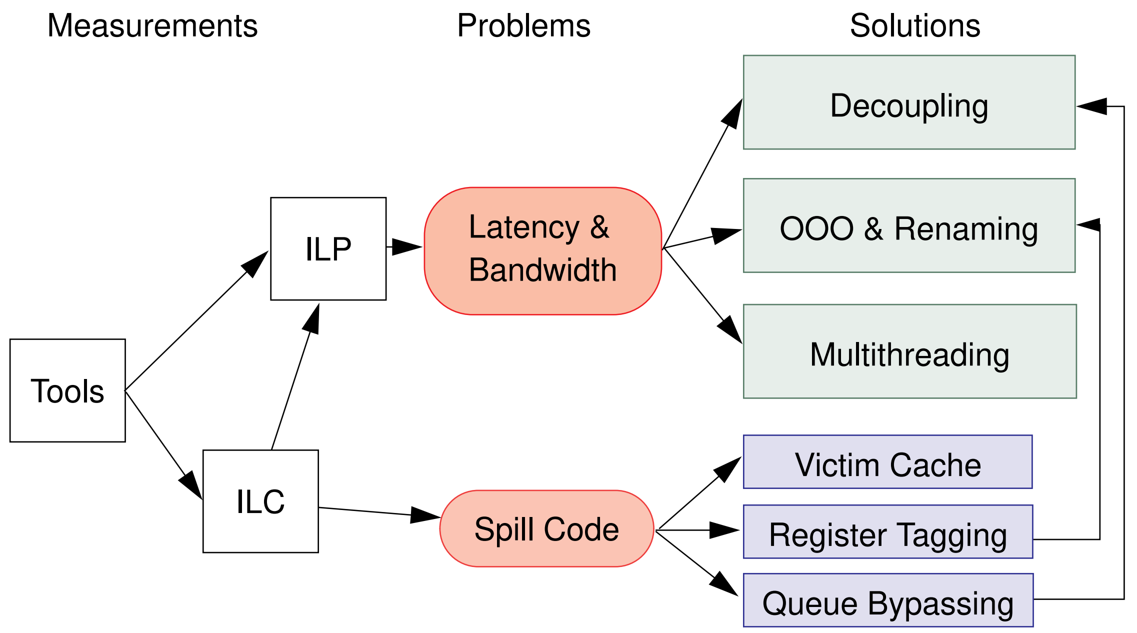

Figure 1.2 Outline of the topics presented in this thesis.

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.