# Multithreaded Vector Architectures

Roger Espasa Mateo Valero

Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya, Barcelona
e-mail: {roger,mateo}@ac.upc.es
http://www.ac.upc.es/hpc

## Abstract

*The purpose of this paper is to show that multithreading techniques can be applied to a vector processor to greatly increase processor throughput and maximize resource utilization. Using a trace driven approach, we simulate a selection of the Perfect Club and Specfp92 programs and compare their execution time on a conventional vector architecture with a single memory port and on a multithreaded vector architecture. We devote an important part of this paper to study the interaction between multithreading and main memory latency. This paper focuses on maximizing the usage of the memory port, the most expensive resource in typical vector computers. A study of the cost associated with the duplication of the vector register file is also carried out. Overall, multithreading provides for this architecture a performance advantage of more than a factor of 1.4 for realistic memory latencies, and can drive the utilization of the single memory port as high as 95%.*

## 1 Introduction

Recent years have witnessed an increasing gap between processor speed and memory speed, which is due to two main reasons. First, technological improvements in cpu speed have not been matched by similar improvements in memory chips. Second, the instruction level parallelism available in recent processors 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 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 three main types of techniques: caching, decoupling and multithreading (which, sometimes, may appear together). Cache-based superscalar processors reduce the average memory access time by placing the working set of a program in a faster level of the memory hierarchy. Software and hardware techniques such as described in [2, 18] have been devised to *prefetch* data from high levels in the memory hierarchy to lower levels (closer to the cpu) before the data is actually needed. On top of that, program transformations such as loop blocking [27] have proven very useful to fit the working set of a program into the cache.

Decoupled scalar processors [22, 21, 15] 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.

Multithreaded scalar processors [1, 24, 25, 13, 5] 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 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.

Vector machines have traditionally tackled the latency problem by exploiting long vectors using vector instructions. Once a (memory) vector operation is started, it pays for some initial (potentially long) latency, but then it works on a long stream of elements and effectively amortizes this latency across all the elements. In vector multiprocessor systems the memory latency can be quite high due to conflicts in the memory modules and in the interconnection network. Although vector machines have been very successful during many years for certain types of numerical calculations, there is still much room for improvement.

Several studies in recent years [20, 8] show how the performance achieved by vector architectures on real programs is far from the theoretical peak performance of the machine. Functional unit hazards and conflicts in the vector register file ports can make vector processors stall for long periods of time and suffer from the same latency problems as scalar processors. In [8] is shown how 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. Also, work by [19] shows how the memory port usage of Cray Y-MP system is also very low, sustaining an average of 1.0 memory transaction per cycle (where the machine could sustain 3 transactions per cycle on its two load and one store memory ports).

Since the memory system of current vector supercomputers is by far the most expensive part of a whole machine, techniques aimed at increasing its utilization are important even if they incur in an increase in cpu cost. Despite this need to improve the memory performance for vector architectures, it is not possible to apply some of the hardware and software techniques used by scalar processors because these techniques are either too expensive or exhibit a poor performance in a vector context. For example, caches and software pipelining are two techniques that have been studied [14, 16, 23, 17] in the context of vector processors but that have not been proved useful enough to be in widespread use in current vector machines.

Decoupled vector architectures have recently been proposed in [7] as a mean of attacking the latency problem in vector processors. This proposal studied decoupling in the context of single memory port vector architectures. In this context, it reduced the total execution time of vector codes even for very small memory latencies and when increasing this latency up to more realistic values, it managed to almost flat out the performance degradation curve. Nevertheless, decoupling did not manage to fully use the total bandwidth of the memory port, and the bus was idle still for a significant fraction of the total execution time.

The purpose of this paper 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.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the proposed multithreaded architecture. Section 4 presents our methodology and the benchmarks used. Section 5 discusses performance bottlenecks in a baseline vector architecture. Section 6 presents the performance of the multithreaded vector architecture. Section 7 studies variations in memory latency and section 8 looks at the effects of the register file crossbars. Section 9 studies the effect of duplicating the control unit and finally section 10 presents our conclusions and future work.

## 2 Related work

Multithreading for scalar programs has received much attention in recent years [24, 25, 13, 5] and has been found to be generally useful. In this paper we diverge from previous work in three key aspects.

First, we will 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 [3, 12] and there has been a commercial machine – the Fujitsu VP2000 series [26]– 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 [25, 11]. This simple design is possible due to the fact that the control unit of a vector processor is typically idle for many cycles [8] and thus can be shared by several threads without penalizing any of them. In this paper we will focus on the simplest design that can achieve a close to optimal level of performance. Nonetheless, a future line of research includes adding register renaming and increasing the complexity of the decode unit, to simultaneously issue from several threads at the same time.

As a third point, and also a consequence of the large granularity of vector operations, we will see how a very small number of threads is needed on a multithreaded vector machine with a single port to almost saturate this resource. As it will be seen in section 6.2, with only 2 threads we can attain an 85-90% occupation of the memory bus.

Finally, we will also note that the type of thread scheduling used in this paper 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. Studies of other policies are currently underway.

## 3 The Multithreaded Vector Architecture

The multithreaded vector architecture we propose is modeled after a Convex C3400 architecture. The base C3400 architecture (henceforth, the *reference architecture*), consists of a scalar part and an independent vector part. The scalar part executes all instructions that involve scalar registers (A and S registers), and issues a maximum of one instruction per cycle. The vector part consists of two computation units (FU1 and FU2) and one memory accessing unit (LD). 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.

The vector unit has 8 vector registers which hold up to 128 elements of 64 bits each one. This eight vector registers are connected to the functional units through a restricted crossbar. Every two 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 to schedule the vector instructions and allocate the vector registers so that no port conflicts arise. The machine modeled chains vectors from functional units to other functional units and to the store unit. It does not chain memory loads to functional units, however. The real Convex C34 does not chain memory loads to functional units (nor do the Cray-2 and Cray-3). 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 our study schedules vector instructions taking the lack of load chaining into account. Because the modeled machine has two read pointers and one write pointer, all implemented chaining is fully flexible – chaining between two dependent instructions may be initiated regardless of the time the second issues.

The multithreaded version of the reference architecture is shown in figure 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 duplicate 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. We are currently looking at how the size of the vector registers can be reduced and traded off against the number of registers while still maintaining the same levels of performance.

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

### 3.1 Machine Parameters

Table 1 presents the latencies of the various functional units present in the architecture. As it can be seen, for a given unit, the latencies for the vector units are larger than those of the scalar units, except for the case of division and square root.

Memory latency is not shown in the table as it will be varied during this study. The memory system modeled is as follows. We have a single address bus shared by all types of memory transactions (scalar/vector and load/store), and physically separate data busses for sending and receiving data to/from main memory [4].

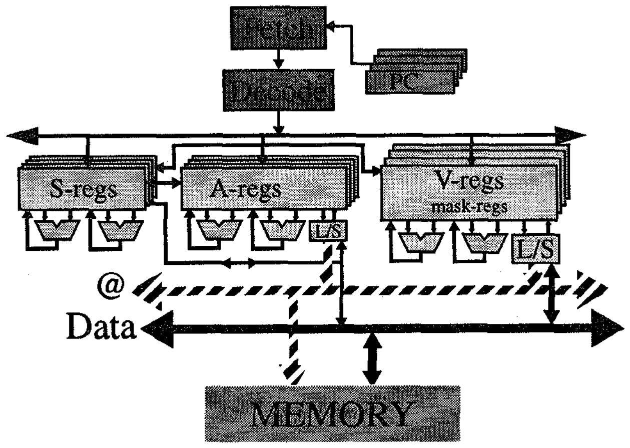

Figure 1: The Multithreaded vector architecture studied in this paper.

| Parameters | Reference | | Multithreaded | |
|---|---|---|---|---|
| | Scal (int/fp) | Vect | Scal (int/fp) | Vect |
| read x-bar | – | 2 | – | 3 |
| write x-bar | – | 2 | – | 3 |
| vector startup | – | 1 | – | 1 |
| add | 1/2 | 6 | 1/2 | 6 |
| mul | 5/2 | 7 | 5/2 | 7 |
| logic/shift | 1/2 | 4 | 1/2 | 4 |
| div | 34/9 | 20 | 34/9 | 20 |
| sqrt | 34/9 | 20 | 34/9 | 20 |

Table 1: Latency parameters for the two architectures studied.

A vector load instruction (also gather instructions) pays an initial latency and then receives one datum per cycle. Vector store instructions do not pay latency since the processor sends the vector to memory and does not wait for the write operation to complete. We will use a value of 50 cycles as the default memory latency. Section 7 will present results on the effects of varying this value.

## 4 Methodology

### 4.1 Simulation Environment

To asses the performance benefits of multithreaded vector architectures we have taken a trace driven approach. A subset of the Perfect Club and Specfp92 programs have been chosen as our benchmarks [9, 10]. These programs are compiled on a Convex C3480 [4] machine and using Dixie [6] a detailed trace that describes its full execution is produced. The tracing procedure is as follows (see figure 2): the benchmark programs are compiled on a Convex C34 machine using the Fortran compiler (version 8.0) at optimization level -O2 (which implies scalar optimizations plus vectorization). Then the executables are processed using Dixie, a tool that decomposes executables into basic blocks and then instruments the basic blocks to produce four types of traces: a basic block trace, a trace of all values set into the *vector length* register, a trace of all values set into the *vector stride* register and a trace of all memory references (actually, a trace of the base address of all memory references). 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. This 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. This runs produce the desired set of traces that accurately represent the execution of the programs. This trace is then fed to two different simulators that we have developed: the first simulator is a model of the Convex C34 architecture and is representative of single memory port vector computers. The second simulator is an extension of the first, where we introduce multithreading. Using these two cycle-by-cycle simulators, we gather all the data necessary to discuss the performance benefits of multithreading.

Our benchmark programs (discussed in 4.2) 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

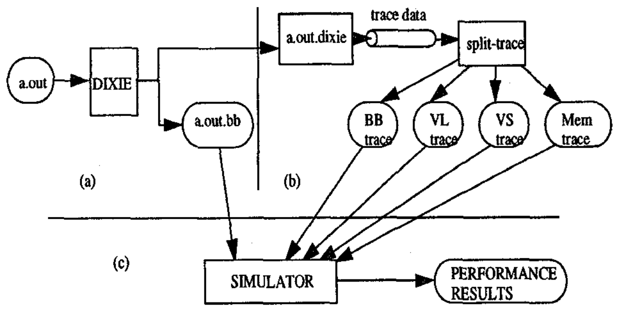

Figure 2: The instrumentation process. Step (a) consists in processing a program's executable and generating an instrumented version of it. In step (b) we run the modified executable on the C3 machine and we obtain a set of traces that fully describe the execution of the program. In (c) this set of traces is fed into the simulator, which will do a cycle-by-cycle execution of the program and will gather performance results.

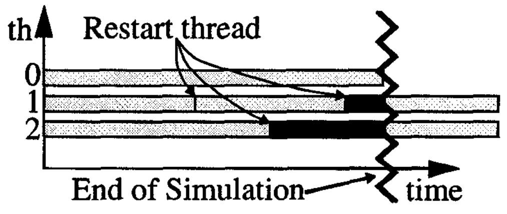

Figure 3: An example run of a multithreaded simulation with 3 hardware contexts.

| Prog. | Num. Threads | | |
|---|---|---|---|
| | 2 | 3 | 4 |
| X | +na<br>+su<br>+to<br>+sw<br>+tf | +sw<br>+hy | +sr |

Table 2: Randomly selected programs to form groupings.

| Program | Suite | #insns S | #insns V | #ops V | % Vect | avg. VL |
|---|---|---|---|---|---|---|
| swm256 (sw) | Spec | 6.2 | 74.5 | 9534.3 | 99.9 | 127 |
| hydro2d (hy) | Spec | 41.5 | 39.2 | 3973.8 | 99.0 | 101 |
| arc2d (sr) | Perf. | 63.3 | 42.9 | 4086.5 | 98.5 | 95 |
| flo52 (tf) | Perf. | 37.7 | 22.8 | 1242.0 | 97.1 | 54 |
| nasa7 (a7) | Spec | 152.4 | 67.3 | 3911.9 | 96.2 | 58 |
| su2cor (su) | Spec | 152.6 | 26.8 | 3356.8 | 95.7 | 125 |
| tomcatv (to) | Spec | 125.8 | 7.2 | 916.8 | 87.9 | 127 |
| bdna (na) | Perf. | 23.9 | 19.6 | 1589.9 | 86.9 | 81 |
| trfd (ti) | Perf. | 352.2 | 49.5 | 1095.3 | 75.7 | 22 |
| dyfesm (sd) | Perf. | 236.1 | 33.0 | 696.2 | 74.7 | 21 |

Table 3: Basic operation counts for the Perfect Club and Specfp92 programs (Columns 2–5 are in millions).

is run to completion at least once, we have taken the following approach (see figure 3): when a group of programs is run, the simulator 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 figure 3, 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

$$(C_0 + 2 \times C_1 + C_2 + F_1 + F_2)/T \qquad (1)$$

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

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 2), 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 2 (example: X+tf); 10 simulations grouping program X with all 5 programs from column 2 and the 2 programs in column 3 (example: X+na+sw); and, finally, 10 simulations grouping program X with all 5 programs from column 2, the 2 programs from column 3 and the program in column 4 (example: X+to+hy+sr).

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

## 4.2 The benchmark programs

Table 3 presents some basic facts about the selected Perfect Club and Specfp92 programs. First column in this table indicates the suite to which each program belongs. Next two columns present the total number of instructions issued by the dispatch unit, broken down into scalar and vector instructions. Column four presents the number of operations performed by the vector instructions. Each vector instruction can perform several operations (up to 128 in the baseline machine), hence the distinction between vector instructions and vector operations. 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 five divided by the sum of columns three and four). Finally 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).

One important point is that we want to evaluate the effects of multithreading for vector programs. Multithreading for scalar programs has been studied extensively in recent years [24, 25, 13, 5]. Our simulations are oriented towards seeing if highly vectorized programs can also benefit from multithreading or not. Therefore, we require from the benchmark programs to be highly vectorizable ($\geq$ 70%) in order to render our results meaningful. From all programs in the Perfect and Specfp92 benchmarks we have selected the top 10 programs meeting our requirement.

## 5 Bottlenecks in the Reference Architecture

This section will present an analysis of the execution of the ten benchmark programs when run through the non-multithreaded architecture simulator.

Consider only the three vector functional units of our reference architecture (FU2, FU1 and LD). The machine state can be represented with a 3-tuple that represents the individual state of each one of the three units at a given point in time. 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.

Figure 4 presents the execution time of the ten benchmark programs broken down into the eight possible states. For each program, we have plotted the execution time for four different values of memory latency. From this figure we can see that the fraction of cycles where these programs proceed at peak floating point speed (states $\langle FU2, FU1, LD\rangle$ and $\langle FU2, FU1,\ \rangle$) is not very high, and that it decreases as memory latency increases. Memory latency has a high impact on total execution time for programs DYFESM, TRFD and FLO52, which have relatively small vector lengths. The effect of memory latency can be seen by noting the increase in cycles spent in state $\langle\ ,\ ,\ \rangle$.

The sum of cycles corresponding to states where the LD unit is idle is quite high in all programs. These four states ($\langle\ ,\ ,\ \rangle$, $\langle\ , FU1,\ \rangle$, $\langle FU2,\ ,\ \rangle$and $\langle FU2, FU1,\ \rangle$) correspond to cycles where the memory port is idle and could (and should) be used to start fetching from memory the data that will be needed by the vector computations in the near future. Figure 5 presents the percentage of these cycles over total execution time. At latency 70, the idle time ranges between a 30% and a 65% of all execution. All those empty memory cycles represent a clear opportunity for executing load/store instructions from another thread. Analysis of the distribution of operations for these programs [7, 8, 19] show that neither the decoding unit nor the arithmetic functional units will be saturated when they run on our reference architecture. This implies that we have "free cycles" where we can try to run other threads in order to saturate the memory port.

## 6 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 4.1. 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.

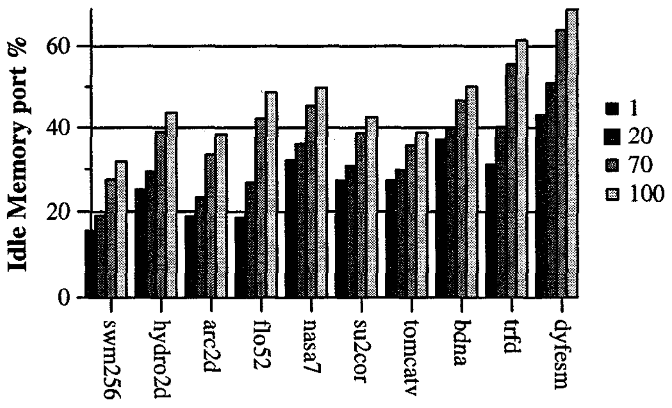

Figure 5: Percentage of cycles where the memory port was idle, for 4 different memory latencies.

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.

As already described in section 4.1, 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 figure 7 is the average memory port utilization computed across all five HYDRO2D simulations (HYDRO2D plus itself, BDNA, SU2COR, TOMCATV and SWM256).

### 6.1 Speedup

The effects of multithreading the reference architecture can be seen in figure 6. 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 4.1.

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.

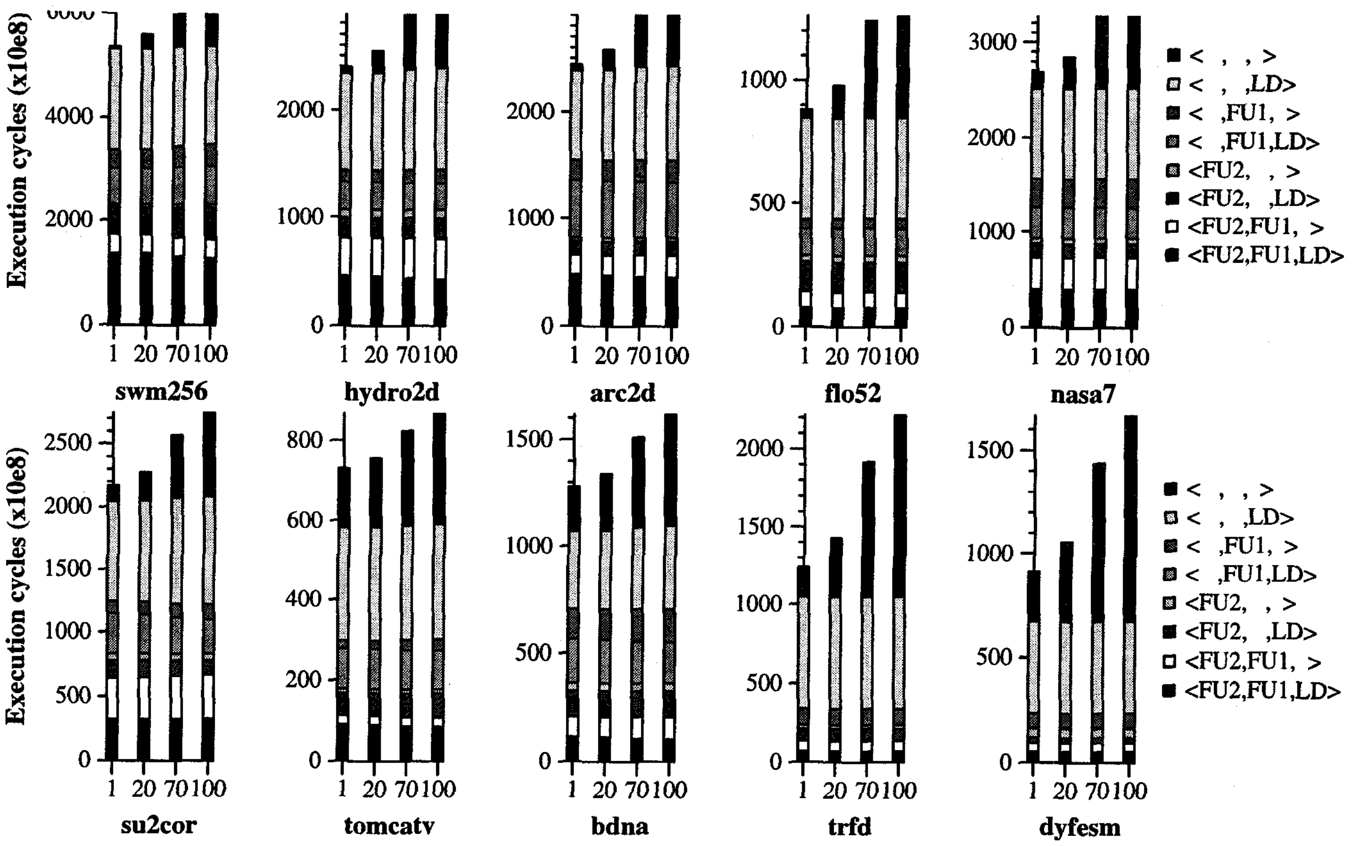

Figure 4: Functional unit usage for the reference architecture. Each bar represents the total execution time of a program for a given latency. Values on the x-axis represent memory latencies in cycles.

## 6.2 Memory Port Usage

The utilization of the single memory port can be seen in figure 7. What we plot in this figure (bars labeled "mth") 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 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 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 baseline 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 figure 7 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.

## 6.3 Vector Operations per cycle

Our third metric of performance for the multithreaded vector architecture is the number of vector

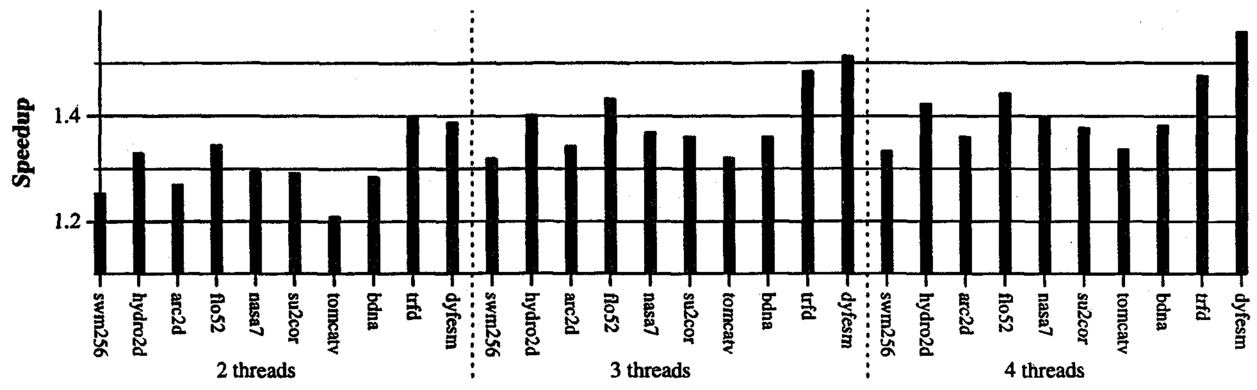

Figure 6: Performance of the Multithreaded approach for 2, 3 and 4 contexts.

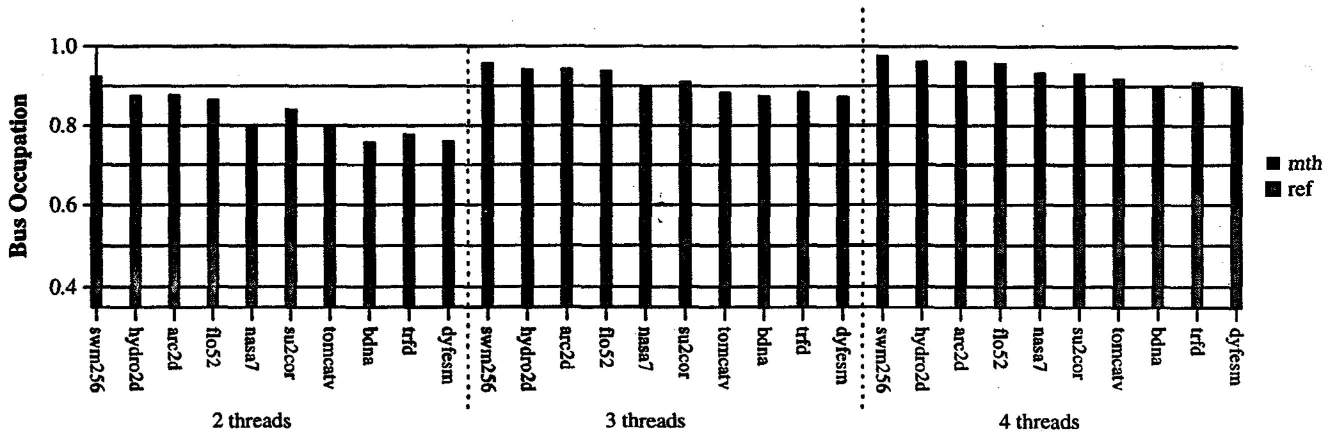

Figure 7: Occupation of the memory port for 2, 3 and 4 contexts for the multithreaded and reference architectures.

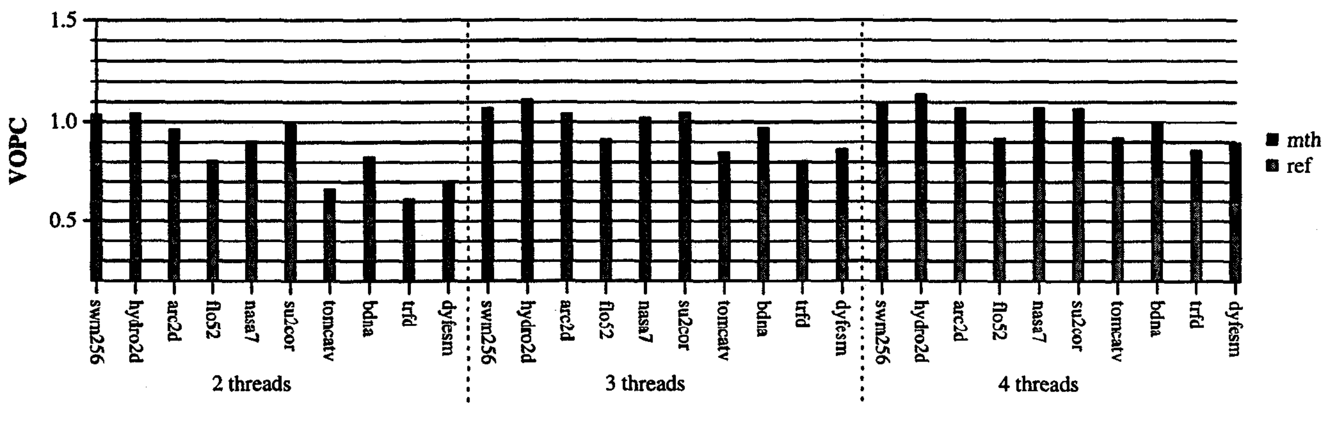

Figure 8: Occupation of the vector functional units for 2, 3 and 4 contexts for the multithreaded and reference architectures.

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 8 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 baseline 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 6.2. If the baseline 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 figure 8 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 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

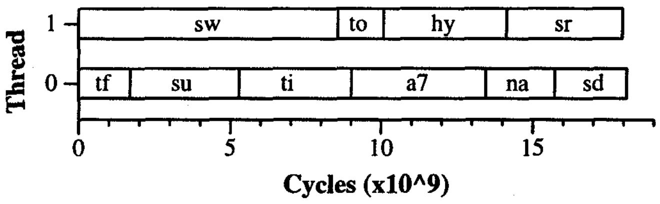

Figure 9: Execution example of the 10 programs run on a 2-context machine with a memory latency value of 50 cycles.

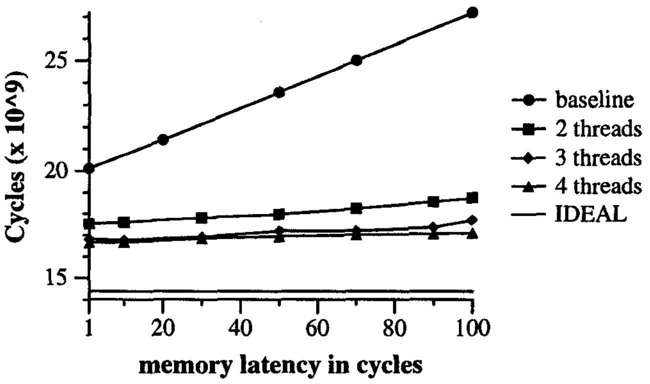

Figure 10: Comparison of total execution time of the 10 benchmarks when run on the baseline machine and on the multithreaded architecture with 2, 3 and 4 contexts, when varying memory latency.

since the two runs have performed slightly different tasks.

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 TF, SW, SU, TI, TO, A7, HY, NA, SR, SD). 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 [13], 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 9 shows an example execution profile of the 10 programs on a 2-context architecture. Note how towards the end, program DYFESM (sd) is for a short period of time alone on the machine.

Figure 10 presents the total execution time of our ten programs when executed on the baseline machine and on the multithreaded machine with 2, 3 and 4 contexts. For the baseline 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 baseline 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 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 baseline 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.

## 8 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 3, duplicating 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 11 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.

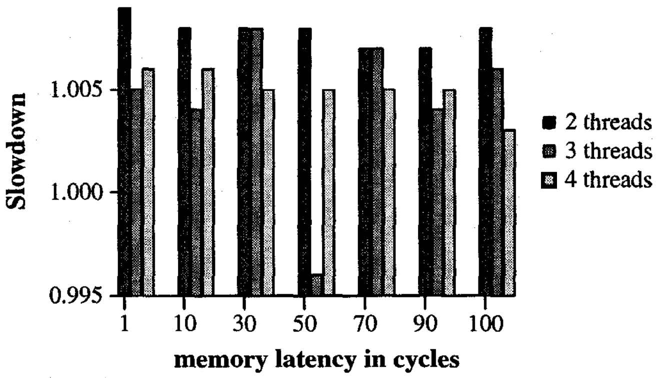

Figure 11: Slowdown due to increasing the latency of the vector registers read/write crossbars from 2 to 3 cycles.

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 (na) and arc2d (sr) interchange their positions from thread 0 to thread 1 and vice-versa. Since our scheduling is biased towards benefiting thread 0, this change in position results in a slightly lower execution time. Thus, the speedup observed is more due to scheduling considerations than to the increase in the crossbars latencies.

## 9 Multithreading versus replicating the scalar processor

As mentioned in section 2, the Fujitsu VP2000 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 [26]). 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,

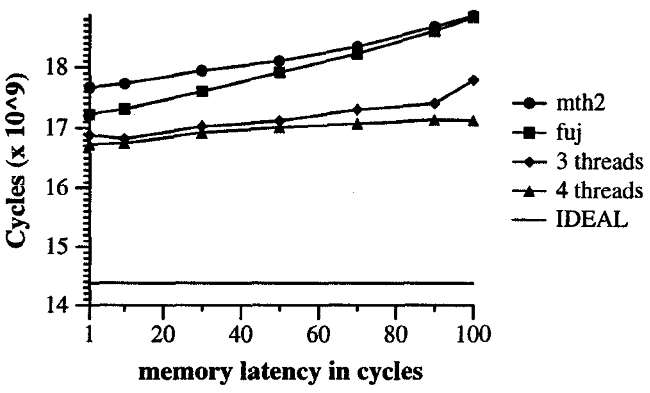

Figure 12: Comparison between a single multithreaded control unit and 2 control units sharing the vector facility.

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 12 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). 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.

## 10 Summary and Future Work

In this paper 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 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%.

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 paper 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.

## References

[1] A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching strategies. In *ISCA*, pages 223–232, 1994.

[3] T. cker Chiueh. Multithreaded vectorization. In *ISCA*, pages 352–361, 1991.

[4] Convex Press, Richardson, Texas, U.S.A. *CONVEX Architecture Reference Manual (C Series)*, sixth edition, April 1992.

[5] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *ISCA*, pages 203–212. ACM Press, May 1996.

[6] R. Espasa and X. Martorell. Dixie: a trace generation system for the C3480. Technical Report CEPBA-RR-94-08, Universitat Politècnica de Catalunya, 1994.

[7] R. Espasa and M. Valero. Decoupled vector architectures. In *HPCA-2*, pages 281–290. IEEE Computer Society Press, Feb 1996.

[8] 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.

[9] M. B. et al. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, pages 5–40, Fall 1989.

[10] R. Giladi and N. Ahituv. SPEC as a performance evaluation measure. *IEEE Computer*, 28(8):33–42, Aug. 1995.

[11] M. Gulati and N. Bagherezadeh. Performance study of a multithreaded superscalar microprocessor. In *HPCA-2*, pages 291–301. IEEE Computer Society Press, Feb 1996.

[12] T. Hashimoto, K. Murakami, T. Hironaka, and H. Yasuura. A micro-vectorprocessor architecture – performance modeling and benchmarking –. In *ICS*, pages 308–317, 1993.

[13] 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 *ISCA*, pages 136–145, 1992.

[14] L. Kontothanassis, R. A. Sugumar, G. J. Faanes, J. E. Smith, and M. L. Scott. Cache performance in vector supercomputers. In *Spercomputing*, 1994.

[15] L. Kurian, P. T. Hulina, and L. D. Coraor. Memory latency effects in decoupled architectures. *IEEE Transactions on Computers*, 43(10):1129–1139, October 1994.

[16] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, June 1988.

[17] W. Mangione-Smith, S. Abraham, and E. Davidson. Vector register design for polycyclic vector scheduling. In *ASPLOS-4*, pages 154–163, Santa Clara, CA, Apr. 1991.

[18] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-5*, 1992.

[19] K. Robbins and S. Robbins. Relationship between average and real memory behavior. *The Journal of Supercomputing*, 8(3):209–232, November 1994.

[20] W. Schonauer and H. Hafner. Explaining the gap between theoretical peak performance and real performance for supercomputer architectures. *Scientific Programming*, 3:157–168, 1994.

[21] J. E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2:289–308, November 1984.

[22] J. E. Smith, S. Weiss, and N. Y. Pang. A simulation study of decoupled architecture computers. *IEEE Transactions on Computers*, C-35(8):692–702, August 1986.

[23] J. Tang, E. S. Davidson, and J. Tong. Polycyclic vector scheduling vs. chaining on 1-port vector supercomputers. *Supercomputing*, pages 122–129, 1988.

[24] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA*, pages 392–403, 1995.

[25] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 191–202. ACM Press, May 1996.

[26] N. Uchida. FUJITSU VP2000 series supercomputers. *International Journal of High Speed Computing*, 3(3 & 4):169–185, 1991.

[27] M. Wolf and M. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991.