Out-of-Order Vector Architectures

Roger Espasa, Mateo Valero*

Computer Architecture Dept.
U. Politècnica de Catalunya-Barcelona
{roger,mateo}@ac.upc.es
http://www.ac.upc.es/hpc

James E. Smith†

Dept. of Electrical & Computer Engr.
University of Wisconsin-Madison
Madison, WI 53706
jes@ece.wisc.edu

Abstract

Register renaming and out-of-order instruction issue are now commonly used in superscalar processors. These techniques can also be used to significant advantage in vector processors, as this paper shows. Performance is improved and available memory bandwidth is used more effectively. Using a trace driven simulation we compare a conventional vector implementation, based on the Convex C3400, with an out-of-order, register renaming, vector implementation. When the number of physical registers is above 12, out-of-order execution coupled with register renaming provides a speedup of 1.24–1.72 for realistic memory latencies. Out-of-order techniques also tolerate main memory latencies of 100 cycles with a performance degradation less than 6%. The mechanisms used for register renaming and out-of-order issue can be used to support precise interrupts – generally a difficult problem in vector machines. When precise interrupts are implemented, there is typically less than a 10% degradation in performance. A new technique based on register renaming is targeted at dynamically eliminating spill code; this technique is shown to provide an extra speedup ranging between 1.10 and 1.20 while reducing total memory traffic by an average of 15–20%.

1 Introduction

Vector architectures have been used for many years for high performance numerical applications – an area where they still excel. The first vector machines were supercomputers using memory-to-memory operation, but vector machines only became commercially successful with the addition of vector registers in the Cray-1 [12]. Following the Cray-1, a number of vector machines have been designed and sold, from supercomputers with very high vector bandwidths [8] to more modest mini-supercomputers. More recently, the value of vector architectures for desktop applications is being recognized. In particular, many DSP and multimedia applications – graphics, compression, encryption – are very well suited for vector implementation [1]. Also, research focusing on new processor-memory organizations, such as IRAM [10], would also benefit from vector technology.

Studies in recent years [13, 5, 11], however, have shown performance achieved by vector architectures on real programs falls short of what should be achieved by considering available hardware resources. Functional unit hazards and conflicts in the vector register file can make vector processors stall for long periods of time and result in latency problems similar to those in scalar processors. Each time a vector processor stalls and the memory port becomes idle, memory bandwidth goes unused. Furthermore, latency tolerance properties of vectors are lost: the first load instruction at the idle memory port exposes the full memory latency.

These results suggest a need to improve the memory performance in vector architectures. Unfortunately, typical hardware techniques used in scalar processors to improve memory usage and reduce memory latency have not always been useful in vector architectures. For example, data caches have been studied [9, 6]; however, the results are mixed, with performance gain or loss depending on working set sizes and the fraction of non-unit stride memory access. Data caches have not been put into widespread use in vector processors (except to cache scalar data).

Dynamic instruction issue is the preferred solution 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. Vector processors have not generally used dynamic instruction issue (except in one recent design, the NEC SX-4 [14]). The reasons are unclear. Perhaps it has been thought that the inherent latency hiding advantages of vectors are sufficient. Or, 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.

Besides in-order vector instruction issue, traditional

*This work was supported by the Ministry of Education of Spain under contract 0429/95, by CIRIT grant BEAI96/II/124 and by the CEPBA.

†This work was supported in part by NSF Grant MIP-9505853.

vector machines have had a relatively small number of vector registers (8 is typical). 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 a shortcoming. Register renaming, useful for out-of-order issue, can come to the rescue here as well. With register renaming more physical registers are made available, and vector register conflicts are reduced.

Another feature of traditional vector machines is that they have not supported virtual memory – at least not in the fully flexible manner of most modern scalar processors. The primary reason is the difficulty of implementing precise interrupts for page faults – a difficulty that arises from the very high level of concurrency in vector machines. Once again, features for implementing dynamic instruction issue for scalars can be easily adapted to vectors. Register renaming and reorder buffers allow relatively easy recovery of state information after a fault condition has occurred.

In this paper, we show that using out-of-order issue and register renaming techniques in a vector processor, performance can be greatly improved. 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. We also present a new technique aimed at dynamically eliminating redundant loads. Using this technique, memory traffic can be significantly reduced and performance is further increased.

2 Vector Architectures and Implementations

This study is based on a traditional vector processor and numerical applications, primarily because of the maturity of compilers and the availability of benchmarks and simulation tools. We feel that the general conclusions will extend to other vector applications, however. The renaming, out-of-order vector architecture we propose is modeled after a Convex C3400. In this section we describe the base C3400 architecture and implementation (henceforth, the *reference architecture*), and the dynamic out-of-order vector architecture (referred to as *OOOVA*).

2.1 The C3400 Reference Architecture

The Convex C3400 consists of a scalar unit and an independent vector unit. The scalar unit executes all instructions that involve scalar registers (A and S registers), and issues a maximum of one instruction per cycle. The vector unit consists of two computation units (`FU1` and `FU2`) and one memory accessing unit

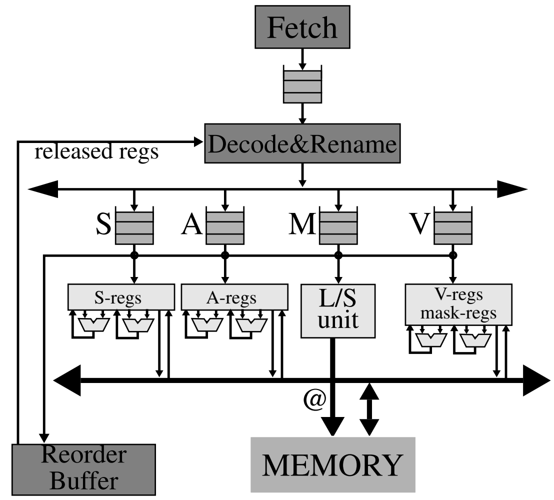

Figure 1: The Out-of-order and renaming version of the reference vector architecture.

(`MEM`). 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. 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 reference machine 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.

2.2 The Dynamic Out-of-Order Vector Architecture (OOOVA)

The out-of-order and renaming version of the reference architecture, OOOVA, is shown in figure 1. It is derived from the reference architecture by applying a renaming technique very similar to that found in the R10000 [16]. Instructions flow in-order through the Fetch 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 register: 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

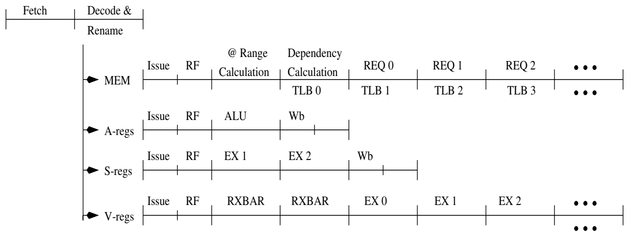

Figure 2: The Out-of-order and renaming main instruction pipelines.

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 only holds a few bits to identify instructions and register names; it never holds register values.

Main Pipelines

There are four main pipelines in the OOOVA architecture (see fig. 2), one for each type of instruction. After decoding and renaming, instructions wait in the four queues shown in fig. 1. 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).

At the Range stage, the range of all addresses potentially modified by a memory instruction is computed. 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), where VL is the vector length register and VS is the vector stride register. Note that the multiplier can be simplified because $VL-1$ is short (never more than 7 bits), 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

Table 1 presents the latencies of the various functional units present in the architecture. Memory latency is not shown in the table because it will be varied. The memory system is modeled as follows. There is 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. Vector load instructions (and gather instructions) pay an initial latency and then receive one datum from memory per cycle. Vector store instructions do not result in observed latency We use a value of 50 cycles as the default memory latency. Section 4.3 will present results on the effects of varying this value.

Parameters	Latency	
	Scal (int/fp)	Vect (int/fp)
read x-bar	–	2
write x-bar	–	2
vector startup	–	(*)
add	1/2	1/2
mul	5/2	5/2
logic/shift	1/2	1/2
div	34/9	34/9
sqrt	34/9	34/9

Table 1: Functional unit latencies (in cycles) for the two architectures.((*) 0 in OOOVA, 1 in REF)

The V register read/write ports have been modified from the original C34 scheme. 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 because renaming shuffles all the compiler scheduled read/write ports and, therefore, would induce a lot of port conflicts.

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. The fetch stage, the decode stage 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.

Commit Strategy

For V registers we start with an aggressive implementation where physical registers are released at the time the vector instruction *begins* execution. Consider the vector instruction: `add v0,v1-->v3`. At the rename stage, v3 will be re-mapped to, say, physical register 9 (ph9), and the old mapping of v3, which was, say, physical 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. 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 writ-

Program	Suite	#insns		#ops	%	avg.
		S	V	V	Vect	VL
swm256	Spec	6.2	74.5	9534.3	99.9	127
hydro2d	Spec	41.5	39.2	3973.8	99.0	101
arc2d	Perf.	63.3	42.9	4086.5	98.5	95
flo52	Perf.	37.7	22.8	1242.0	97.1	54
nasa7	Spec	152.4	67.3	3911.9	96.2	58
su2cor	Spec	152.6	26.8	3356.8	95.7	125
tomcatv	Spec	125.8	7.2	916.8	87.9	127
bdna	Perf.	239.0	19.6	1589.9	86.9	81
trfd	Perf.	352.2	49.5	1095.3	75.7	22
dyfesm	Perf.	236.1	33.0	696.2	74.7	21

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

ing 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 5 will change the release algorithm to allow for precise exceptions.

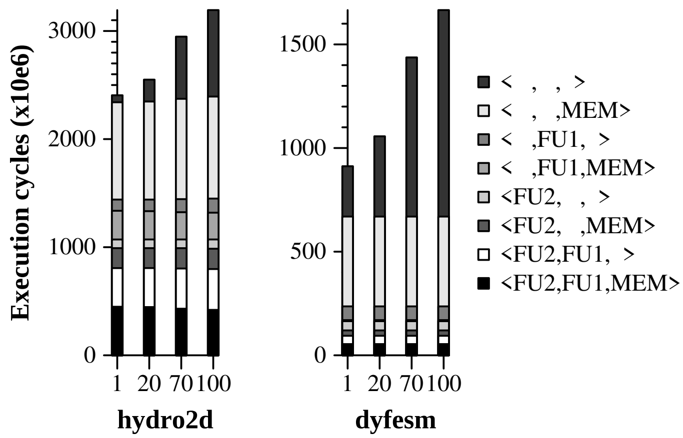

Figure 3: Functional unit usage for the reference architecture. Each bar represents the total execution time of a program for a given latency. Values on the x-axis represent memory latencies in cycles.

3 Methodology

To assess the performance benefits of out-of-order issue and renaming in vector architectures we have taken a trace driven approach. A subset of the Perfect Club and Specfp92 programs is used as the benchmark set. These programs are compiled on a Convex C3480 machine and the tool Dixie [3] is used to modify the executable for tracing. 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 simulators for the reference and OOOVA architectures.

3.1 The benchmark programs

Because we are interested in the benefits of out-of-order issue for vector instructions, we selected benchmark programs that are highly vectorizable. From all programs in the Perfect and Specfp92 benchmarks we chose the 10 programs that achieve at least 70% vectorization. Table 2 presents some statistics for the selected Perfect Club and Specfp92 programs. Column number 2 indicates to what suite each program belongs. Next two columns present the total number of instructions issued by the decode unit, broken down into scalar and vector instructions. Column five presents the number of operations performed by vector instructions. The sixth column is the percentage of vectorization of each program (i.e., column five divided by the sum of columns three and five). Finally, column seven presents the average vector length used by vector instructions (the ratio of columns five and four, respectively).

4 Performance Results

4.1 Bottlenecks in the Reference Architecture

First we present an analysis of the execution of the ten benchmark programs when run through the reference architecture simulator.

Consider the three vector functional units of the reference architecture (`FU2`, `FU1` and `MEM`). The machine state can be represented with a 3-tuple that captures the individual state of each of the three units at a given point in time. For example, the 3-tuple $\langle FU2, FU1, MEM\rangle$ represents a state where all units are working, while $\langle\ ,\ ,\ \rangle$ represents a state where all vector units are idle.

Figure 3 presents the execution time for two of the ten benchmark programs (see [4] for the other 8 programs). Space limitations prevents us from providing them all, but these two, hydro2d and dyfesm, are representative. During an execution the programs are in eight possible states. 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 $\langle FU2, FU1, MEM\rangle$ 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. Memory latency has a high impact on total execution time for programs dyfesm (shown in Figure 3), and trfd and flo52 (not shown), 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 MEM 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 mem-

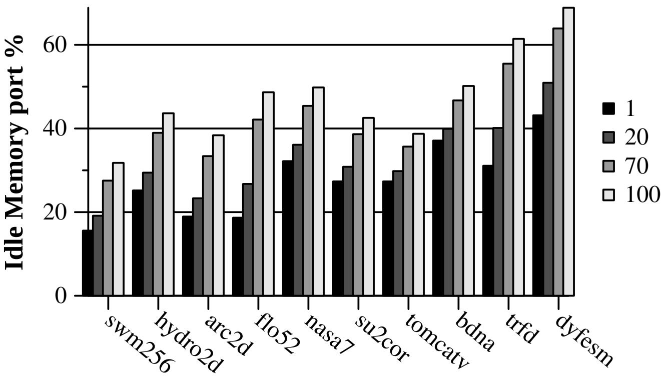

Figure 4: Percentage of cycles where the memory port was idle, for 4 different memory latencies.

ory port could potentially be used to fetch data from memory for future vector computations. Figure 4 presents the percentage of these cycles over total execution time. At latency 70, the port idle time ranges between 30% and 65% of total execution time. All 10 benchmark programs are memory bound when run on a single port vector machine with two functional units. Therefore, these unused memory cycles are not the result of a lack of load/store work to be done.

4.2 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 figure 5. 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 top of each graph). To compute the IDEAL speedup for a program we use the total number of cycles consumed by the most heavily used vector unit (FU1, FU2, or MEM). 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.

As can be seen from figure 5, 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. 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 figure 6. 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.

Resource Usage

We now consider resource usage for the OOOVA machine and compare it with the reference machine. This is illustrated in figure 7. The same notation as in figure 3 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 7 shows that the major improvement is in state $\langle\ ,\ ,\ \rangle$, which has almost disappeared. Also, the fully-utilized state, $\langle FU2, FU1, MEM\rangle$, 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.

4.3 Tolerance of Memory Latencies

One way of looking at the advantage of out-of-order execution and register renaming is that it allows long

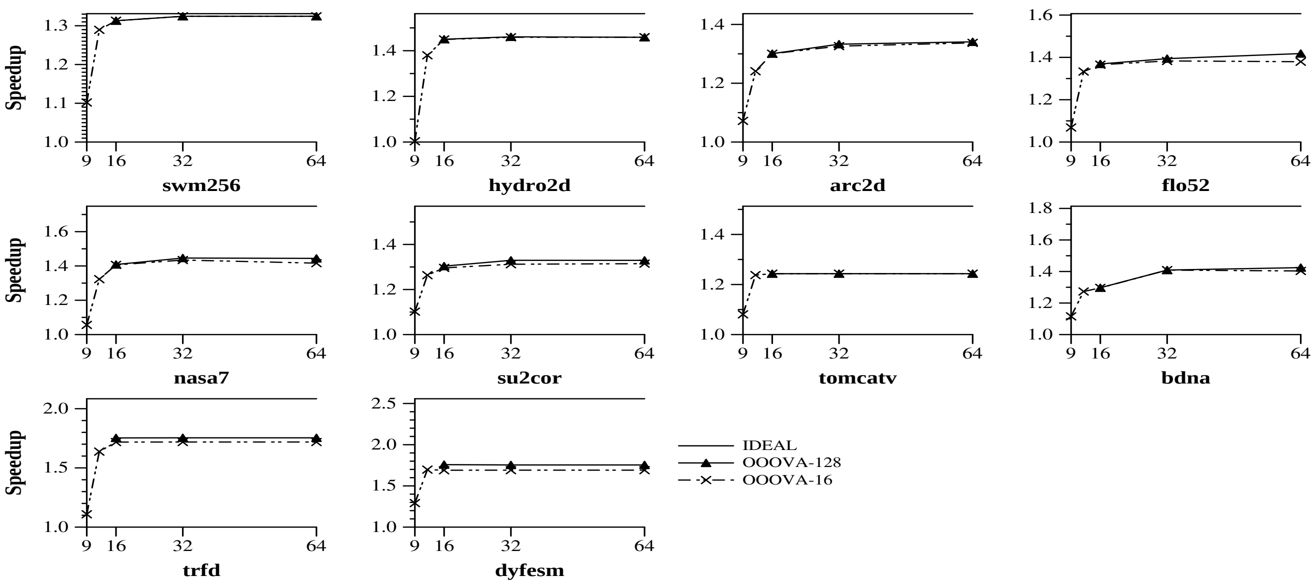

Figure 5: Speedup of the OOOVA over the REF architecture for different numbers of vector physical registers.

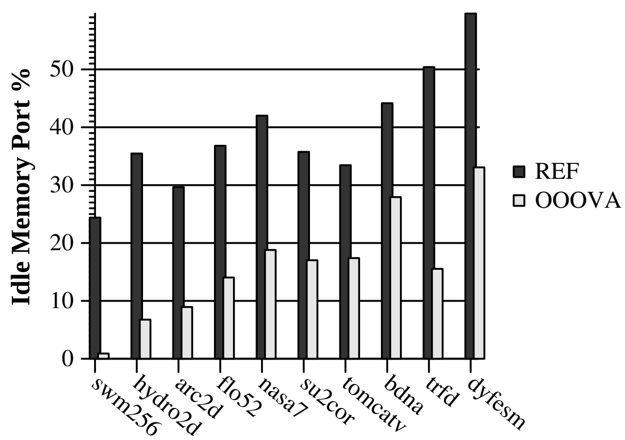

Figure 6: Percentage of idle cycles in the memory port for the Reference architecture and the OOOVA architecture. Memory latency is 50 cycles and the vector register file holds 16 physical vector registers.

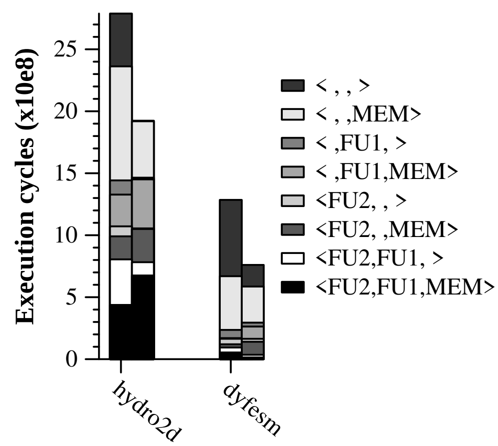

Figure 7: Breakdown of the execution cycles for the REF (left bar) and OOOVA (right bar) machines. The OOOVA machine has 16 physical vector registers. For both architectures, memory latency was set at 50 cycles.

memory latencies to be hidden. In previous subsections we showed the benefits 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 8 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 of the increase in memory latency. For most 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.

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 individ-

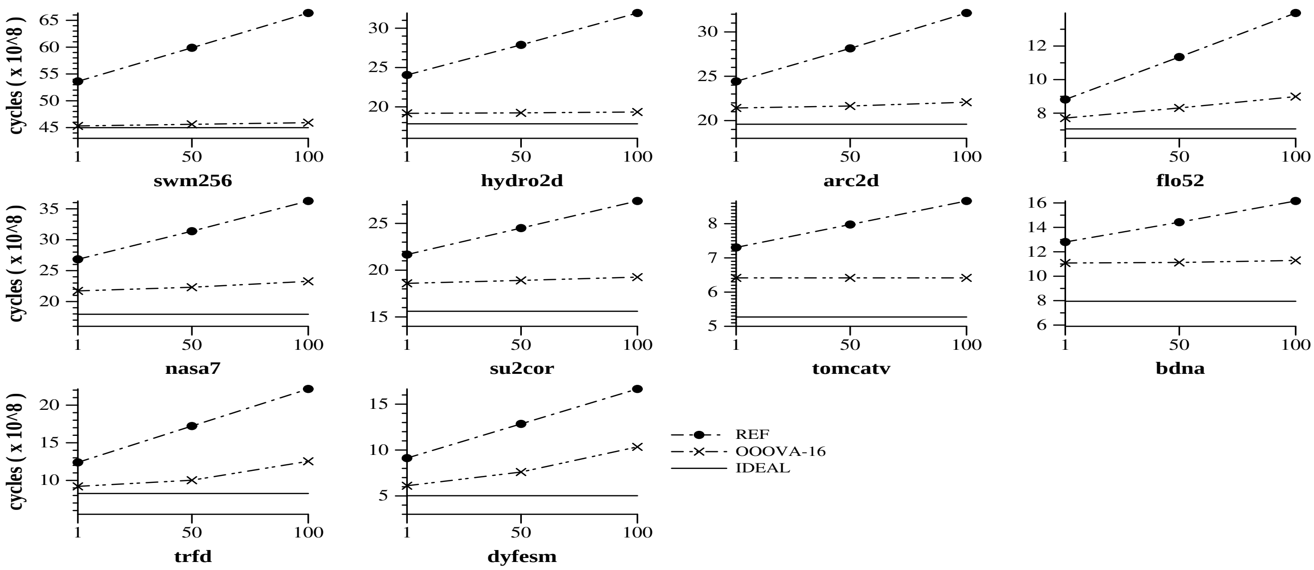

Figure 8: Effects of varying main memory latency for three memory models and for the 16 physical vector registers machines.

ual memory modules in the memory 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.

5 Implementing Precise Traps

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 [15], but is not used in many current high performance parallel vector processors [7]. Or, it is used in a very restricted form, for example by locking pages containing vector data in memory while a vector program executes [7, 14].

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 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 traps, 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.

Figure 9 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 two important observations about the graphs in Figure 9. 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. 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

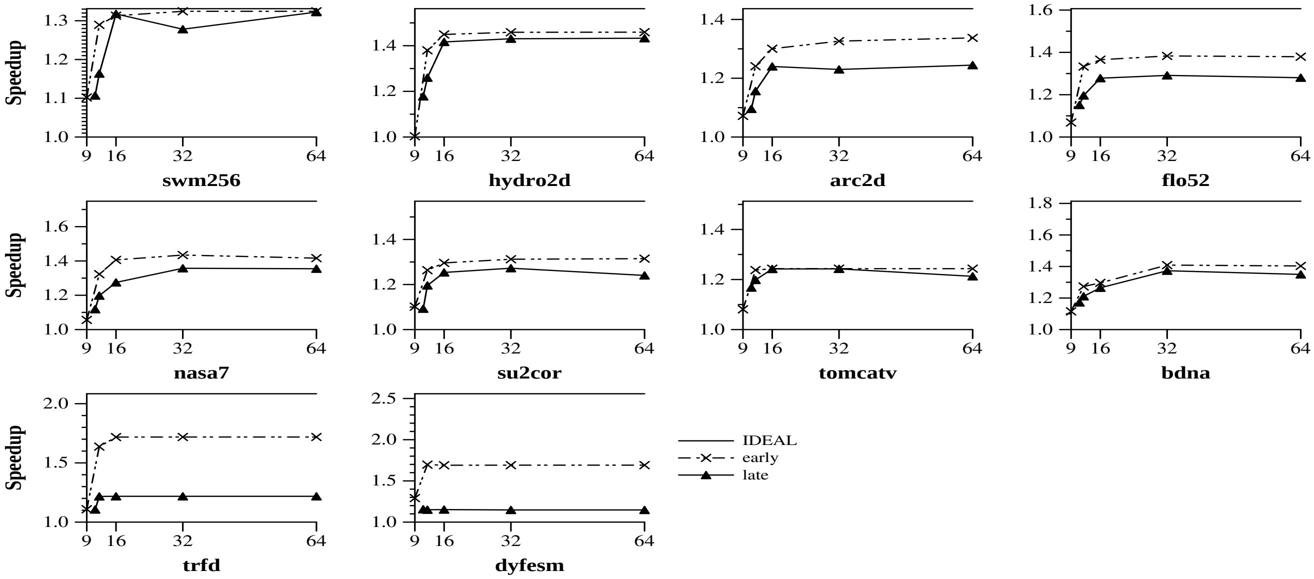

Figure 9: Speedups of the OOOVA over the reference architecture for different numbers of vector physical registers under the early and late commit schemes.

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.

6 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.

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. To illustrate the importance of spill code for vector architectures, table 3 shows the number of memory spill operations (number of words moved) in the ten benchmark programs. In some of the benchmarks relatively few of the loads and stores are due to spills, but in several there is a large amount of spill traffic. For example, over 69% of the memory traffic in bdna is due to spills.

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

Program	Vector load ops			Vector store ops			Total
	load	spill	%	store	spill	%	%
swm256	2839	315	10	1030	315	23	14
hydro2d	1297	21	1.6	431	21	5	2.4
arc2d	1244	122	9	479	87	15	11
flo52	428	41	8.8	181	41	19	12
nasa7	1048	21	2.0	632	20	3	2.4
su2cor	786	201	20	404	103	20	20
tomcatv	234	104	31	72	104	59	41
bdna	142	266	65	71	221	76	69
trfd	433	0	0	224	0	0	0
dyfesm	289	0.5	0.2	108	0.5	0.4	0.2

Table 3: Vector memory spill operations. Columns 2, 3, 5 and 6 are in millions of operations.

load for spilled data is executed in nearly zero time. We do not eliminate spill stores, however, because of the need to maintain strict binary compatibility. That is, 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.

6.1 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: $\langle @_1, @_2, \mathtt{vl}, \mathtt{vs}, \mathtt{sz}, \mathtt{v} \rangle$. 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. 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.

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. If the operation is a store, then the physical register being stored to memory has its tag updated with the corresponding address information. 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.

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 [2]. 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.

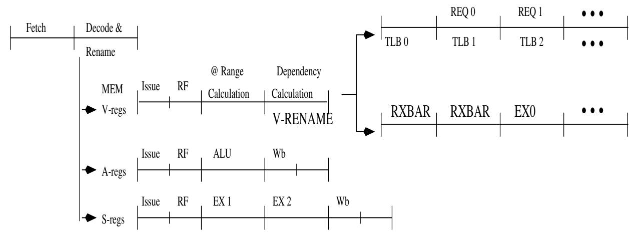

Figure 10: The modified instruction pipelines for the Dynamic Load Elimination OOOVA.

6.2 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 10 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 are available to all following vector instructions. Moreover, this ensures that store tags are compared against all previous tags in order.

6.3 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).

Figures 11 and 12 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 (figure 12), speedups over the base OOOVA are from 1.04 to 1.16 for most programs and are as high as 1.78 and 2.13 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

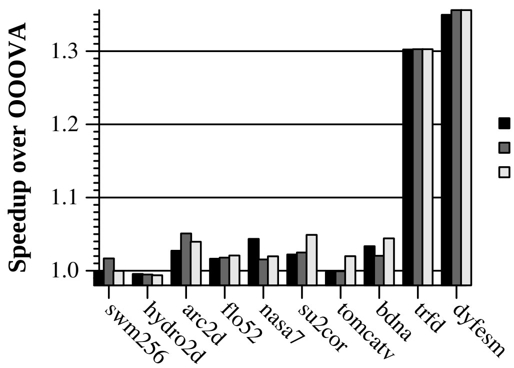

Figure 11: Speedup of SLE over the OOOVA machine for 3 different physical vector register file sizes.

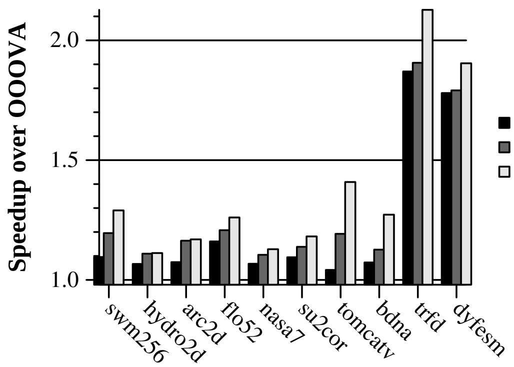

Figure 12: Speedup of SLE+VLE over the OOOVA machine for 3 different physical vector register file sizes.

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 1.19 up to 1.40). 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 (figure 11). 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. In particular, the ability to bypass data between loads and stores allows them to unroll the two most critical loops, whereas without SLE, the unrolling was not possible.

6.4 Traffic Reduction

A very important effect of dynamic load elimination is that it reduces the total amount of traffic seen by the memory system. This is a very important feature

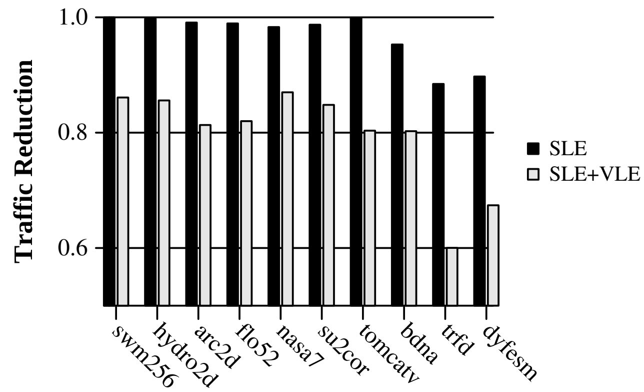

Figure 13: Traffic reduction under dynamic load elimination with 32 physical vector registers.

in multiprocessing environments, where less load on the memory modules usually translates into an overall system performance improvement.

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 baseline OOOVA divided by the total number of requests done by either the SLE or the SLE+VLE configurations. Figure 13 present this ratio for 32 physical vector registers. As an example, figure 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, 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%.

7 Summary

In this paper 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 to and from 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.

References

[1] K. Asanovic, J. Beck, B. Irissou, B. Kingsbury, N. Morgan, and J. Wawrzynek. The T0 Vector Microprocessor. In *Hot Chips VII*, pages 187–196, August 1995.

[2] H. Dietz and C.-H. 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.

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

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

[5] 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.

[6] J. 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.

[7] A. Iwaya and T. Watanabe. The parallel processing feature of the NEC SX-3 supercomputer system. *Intl. Journal of High Speed Computing*, 3(3&4):187–197, 1991.

[8] K. Kitai, T. Isobe, T. Sakakibara, S. Yazawa, Y. Tamaki, Teruo, and K. Ishii. Distributed storage control unit for the Hitachi S-3800 multivector supercomputer. In *ICS*, pages 1–10, July 1994.

[9] L. 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.

[10] D. Patterson, T. Anderson, and K. Yelick. A Case for Intelligent DRAM: IRAM. In *Hot Chips VIII*, August 1996.

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

[12] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

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

[14] P. Tannenbaum. HNSX Supercomputers Inc.; Marketing Group Director, 1996. Private Communication.

[15] T. Utsumi, M. Ikeda, and M. Takamura. Architecture of the VPP500 Parallel Supercomputer. In *Proceedings of Supercomputing'94*, pages 478–487, Washington D.C., November 1994. IEEE Computer Society Press.

[16] K. C. Yager. The Mips R10000 Superscalar Microprocessor. *IEEE Micro*, pages 28–40, April 1996.