Effective Usage of Vector Registers in Advanced Vector Architectures

Luis Villa * Roger Espasa Mateo Valero

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

Abstract

This paper presents data confirming the fact that traditional vector architectures can not reduce their vector register length without suffering a severe performance penalty. However, we will show that by combining the vector register length reduction with two different ILP techniques, decoupling and multithreading, the performance penalty can be made very small. We will show that each resulting architecture tolerates very well long memory latencies and also makes a better usage of the available storage space in each vector register. Using decoupling and short vectors, each register can be halved while still providing speedups in the range 1.04–1.49 over a traditional architecture with long registers. Using multithreading, we split a vector register file in two halfs and show that two independent threads running on such machine can yield speedups in the range 1.23–1.29. The paper also explores configurations with 1/4 and 1/8 the original vector register size aimed at cost-consciouss designs, and shows that even at 1/4 the original size, the resulting architectures can outperform a traidtional machine. We also present results across a wide range of memory latencies, and show that the combination of short vectors and ILP techniques results in a very good tolerance of slow memory systems.

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 [1, 2], but vector machines only became commercially successful with the addition of vector registers in the Cray-1 [3]. Following the Cray-1, a number of vector machines have been designed and sold, from supercomputers with very high vector bandwidths [4, 5] to more modest mini-supercomputers [6, 7].

The traditional approach to vector processor design has been to use an in-order execution engine and achieve high performance exploiting the natural data-level parallelism embedded in each vector instruction. Typically, traditional vector architectures have used very limited forms of ILP techniques, only allowing some overlapping of vector and scalar instructions but keeping the scalar and vector instruction streams strictly ordered. To achieve good performance and to be able to tolerate the large latencies associated with supercomputer main memory systems, vector designers have exploited the large number of independent operations present in each vector instruction. When a vector instruction 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 elements. A few of these vector instructions running concurrently can yield a very good usage of the available hardware resources.

In this context, it is natural that vector processor designers have striven to implement vector registers as large as budget and technology constraints would allow. Nonetheless, in today's environment where ILP techniques such as out-of-order execution, decoupling, multithreading, branch prediction, speculation, etc, have proved their value as latency tolerance mechanisms, it is less clear that the best way to invest the available register space consists in having only few very large registers.

Large registers have several drawbacks. First, if an application can not make full use of each register, then a precious hardware resource is being wasted. Second, given a certain budget in terms of transistors, large registers imply that only a few of them can be implemented. A small number of logical registers has a direct impact on the amount of spill code that the compiler and/or programmer must introduce to fit all live variables in the limited register file. Third, introducing ILP techniques in a processor having a few very large logical registers is difficult. For example, out-of-order execution without renaming with only 8 logical vector registers provides little benefit. On the other hand, introducing register renaming can be very costly since many copies of registers that are very large have to be provided.

*This work was supported by the Ministry of Education of Spain under contract 0429/95, by the Instituto de Cooperacion Iberoamericana (ICI) and by the CEPBA.

Reducing the vector registers length is certainly a solution to the problems just outlined. If most applications can not fully use all elements present in each vector register [8], then reducing the vector register length will reduce cost and increase the fraction of usage of registers. The drawback of register length reduction is the associated performance penalty. Each time a vector instruction is executed, its associated latencies are amortized over a smaller number of elements. This can have a significant impact on performance, especially for memory accesses. Moreover, more instructions have to be executed each with a shorter effective length, and, therefore, the number of times that latencies must be payed is larger.

Unless some extra latency tolerance mechanism is introduced in a vector architecture, vector length can not be reduced without a severe performance penalty. While many techniques have been developed to tolerate memory latency in superscalar processors, only a few studies have considered the same problem in the context of vector architectures [9, 10, 11].

This paper will present data confirming the fact that traditional vector architectures can not reduce their vector register length without suffering a severe performance penalty. However, we will show that by combining the vector register length reduction with two different ILP techniques, decoupling and multithreading, the performance penalty can be made very small. We will show that each resulting architecture tolerates very well long memory latencies and also makes a better usage of the available storage space in each vector register. Not only the performance impact of reducing the vector length is small, but when our two architectures with short vector registers are compared against a traditional vector machine with large vector registers, performance is in most cases far better across a large memory latency range.

2 Vector Length Distributions

The usage of the vector register file elements is determined by both the degree of vectorization of a program and the natural vector lengths associated with the data structures of an application. Many applications have small data sets or iterate over a particular dimension of an iteration space which is smaller than the vector register length. We start by evaluating a set of vectorizable applications to see what is their usage of a traditional vector register file. To select a set of highly vectorizable benchmarks, we compiled all programs from the Perfect Club and Specfp92 suites on a Convex C3400 machine, which has a maximum vector length of 128 elements. Then we selected the ten most vectorizable programs.

Table 1 presents some statistics for the selected 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. Each vector instruction can perform many operations (up to 128), hence the distinction between vector instructions and vector operations. The sixth column is the percentage of vectorization of each program. We define the percentage of vectorization 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 five). Finally column seven presents the average vector length used by vector instructions, and is the ratio of vector operations and vector instructions (columns five and four, respectively).

Program	Suite	#insns S	#insns V	#ops V	% Vect	avg. 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.	23.9	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 1: Basic operation counts for the Perfect Club and Specfp92 programs (Columns 3–5 are in millions).

The first thing to note is that, even though these ten programs are highly vectorizable, their average vector lengths are not very high. Investigation of the programs reveals that often times this is due to the natural shape of the application data space. In other cases, it is due to the nature of the algorithm, i.e., a triangular matrix operation tends to have many small vector lengths.

To better illustrate this point, we monitored every single vector instruction and recorded the vector length that it uses. The accumulated percentage distribution of these values is plotted in figure 1. From figure 1 shows that the vector length distributions do not follow any regular pattern. Four programs behave as expected (`swm256`, `tomcatv` and, to a large extent, `su2cor` and `bdna`), having the majority of their vector lengths clustered around 128 and having a small percentage of very short vector lengths that are the residuals generated due to strip-mining. `Hydro2d` has a single dominant vector length (102) that comes from the fact that this value is the number of grid points used in the z-direction of the problem. Programs `nasa7`, `arc2d` and `flo52` have a distribution that follows a staircase, having several dominant vector lengths. Programs `trfd` and `dyfesm` have almost all their vector lengths lower than 64.

3 Full Stripes

Given the results presented in the previous section, we want to investigate how the applications vector

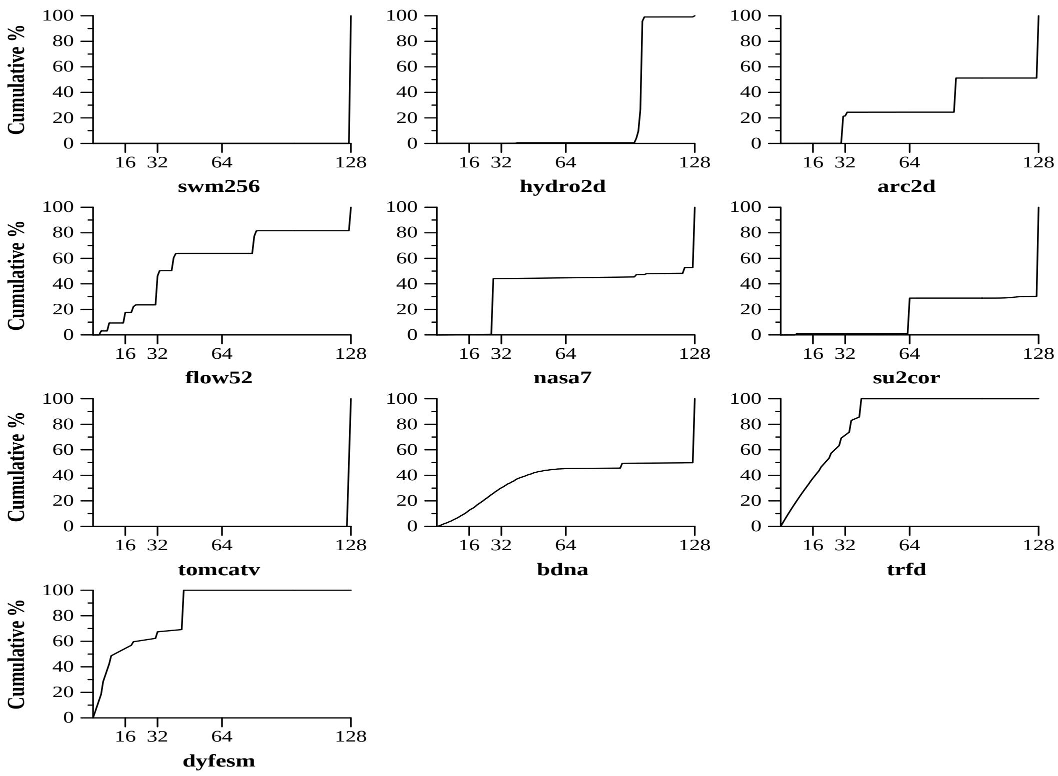

Figure 1: Cumulative percentage distribution of the vector lengths used during program execution.

length and the hardware vector register length are related. That is, if we vary the vector register size, how many times will we have a vector register completely filled with data ? What we have done is compute the percentage of times that a vector register is used in all its capacity (we term this situation as a "full stripe"). For example, say a program has two vector instructions, and that the first instruction uses a vector length of 128 and the second one a vector length of 72. If we consider the vector register size to be 128, we have that 50% of all instructions have used a full stripe. Now, if we consider a vector register size of 64 elements the first instruction would "translate" into two instructions that would operate on 64 elements each one. The second instruction would turn into two instructions also, but that would operate on 64 and 8 elements respectively. Thus, 3 out of 4 instructions (75%) would be using a full stripe.

Figure 2 presents the percentage of full stripes of each program, considering four possible values for the vector register size: 16, 32, 64 and 128 elements. For example, the figure shows how in BDNA 50% of all executed vector instructions used a vector register of size 128. For the same program, if the vector register size was 16 elements, almost 92% of all vector instructions would fully use the vector registers.

From figure 2 we can see that in order to augment the percentage of full stripes we would have to choose a relatively small vector register size. Next sections will look into the performance implications of choosing a small vector register size.

4 Compiling for smaller vector lengths

In order to investigate the effects of reducing the hardware vector register length we need a set of benchmarks compiled assuming different vector lengths. Unfortunately, no public domain vectorizing compiler is available and, therefore, we are forced to artificially fool the Convex compiler [6] to generate code "as if" the vector length was 16, 32 or 64 (instead of the real 128). To obtain the desired binaries we modified the source benchmarks as follows. Using the vectorization information produced by the Convex compiler, we located in the source code each vectorized loop. For each loop nest, and taking into account loop transformations such as peeling, interchange and skewing, we manually strip-mined the loop being vectorized. This manual strip-mining consisted in adding a strip mine loop performing steps of length VLZ and modifying the original vectorized loop to do at most VLZ iterations (see figure 3). To prevent the compiler from generating a doubly strip-mined loop (our strip-mining plus the natural strip mining introduced by the compiler) we used the MAXTRIPS directive [6]. This directive informed the compiler that the inner loop was performing less than 128 trips and thus no extra strip-mining was generated.

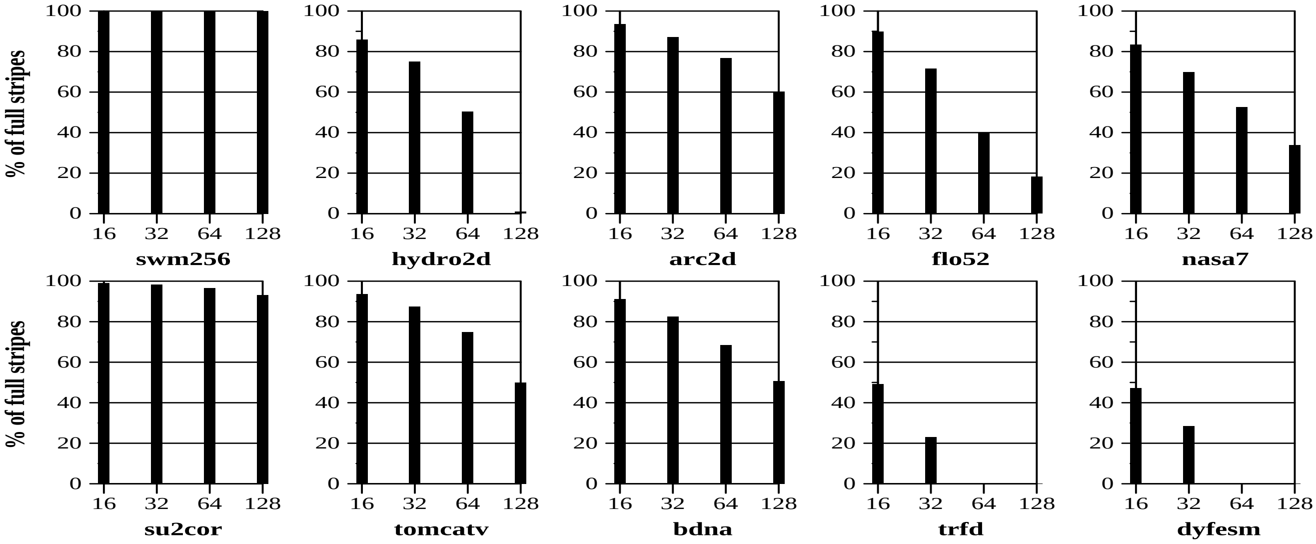

Figure 2: Percentage of full stripes for different vector register sizes

```
DO 40 J=2,JL
    DO 40 I=2,IL
    DW(I,J,1) = DW(I,J,1)  +FW(I,J,1)
    DW(I,J,2) = DW(I,J,2)  +FW(I,J,1)
    DW(I,J,3) = DW(I,J,3)  +FW(I,J,3)
    DW(I,J,4) = DW(I,J,4)  +FW(I,J,4)
40 CONTINUE
                    (a)
DO 40 J=2,JL
    DO 40 STRIPV=2,IL,VLZ
C$DIR MAX_TRIPS(VLZ)
    DO 40 I=STRIPV,MIN(IL,STRIPV+VLZ)
    DW(I,J,1) = DW(I,J,1)  +FW(I,J,1)
    DW(I,J,2) = DW(I,J,2)  +FW(I,J,2)
    DW(I,J,3) = DW(I,J,3)  +FW(I,J,3)
    DW(I,J,4) = DW(I,J,4)  +FW(I,J,4)
40 CONTINUE
                    (b)
```

Figure 3: (a) `Flo52` loop without Strip-Mining, (b) Adding Strip-mining.

Using such a procedure we strip-mined most (but not all) vectorized loops present in our ten benchmarks. Loops that escaped from this strip-mining where vector loops that are in libraries and loops where introducing one extra level of strip-mining stopped vectorization. Moreover, due to the large number of loops to strip-mine, we first selected those that accumulate 95% of all execution time. The remaining loops that form the other 5% of execution time were not instrumented. For each program, we generated four different binaries, assuming that the

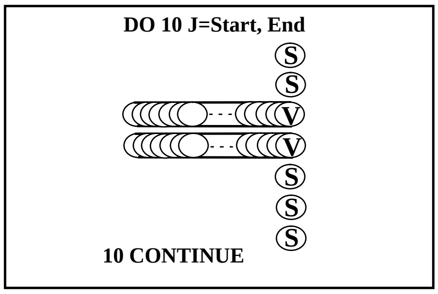

Figure 4: Typical body loop, executed in a Vector Architecture.

maximum hardware vector length was 16, 32, 64 and 128. For each register length, the percentage of operations that escaped our strip-mining procedure varied, but was below 4% for all programs except `arc2d` and `flo52` where it was close to 10%.

A very important effect of reducing the hardware vector register length is that the total number of vector and scalar instructions executed varies dramatically. Consider a very simple loop as shown in figure 4. It has 5 scalar instructions and 2 vector instructions. This loop copies 256 data items from array A into array B. Assuming a register length of 128, this task can be performed in just 2 iterations, yielding a total of 5×2 scalar operations being executed. On the other hand, assuming a register length of 16, the loop requires 8 iterations and executes 40 scalar instructions. Also, more vector instructions have been executed (16 versus 4). Translating this into vectorization percentages, the first loop executes 512 operations in vector mode and 10 in scalar mode, yielding a $512/(512{+}10) = 98\%$ vectorization. The second loop also performs 512 operations in vector mode but now performs 40 scalar operations, yielding a $512/(512+40) = 92.7\%$ vector-

Programs	% of operations Vectorized			
	128	64	32	16
SWM256	99.5	99.1	98.3	96.8
HYDRO2D	96.5	94.7	91.2	86.4
ARC2D	96.6	95.3	93.2	88.4
FLOW52Q	92.0	90.8	88.5	83.8
NASA7	94.0	92.1	88.9	82.4
SU2COR	88.4	87.0	84.5	80.0
TOMCATV	87.0	86.5	85.8	83.5
BDNA	86.8	86.4	85.7	84.3
TRFD	67.8	67.7	65.5	60.8
DYFESM	65.7	65.3	63.8	58.8

Table 2: Degradation of vectorization percentage for different vector lengths.

ization.

Table 4 shows this effect for all benchmark programs. As the vector register length is reduced, the vectorization percentage decreases. It is worth noting that this decrease is not linear. It depends on the distribution of vector lengths used in the original, non strip-mined, program.

5 Short Vectors Performance

We start by analyzing the performance of a traditional in-order vector machine when the hardware vector length is varied. We are interested in the effect that different memory latencies have on performance and how it interacts with vector register length.

5.1 Architecture

Our reference machine is loosely based on a Convex C3400. The Convex C3400 [6] 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 (`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.

We will study four different variants of this reference machine. Each variant will have a particular vector register length. The four models under study will be referred to as the REF128, REF64, REF32 and REF16 architectures and will have a vector length of 128, 64, 32 and 16 elements respectively.

5.2 Performance

Figure 5 plots the performance of each program under different register lengths and different memory latencies on the reference machine. For each program we plot the slowdown with respect to a baseline machine having 128 as its vector length and using a 1 cycle main memory latency.

The impact of memory latency can be clearly seen in figure 5. If we focus on the unmodified, 128-length machine (curve REF128) we can see that execution time is degraded by factors of 1.2–1.4 in most programs and for `dyfesm` and `trfd` the slowdown can be as high as 1.7. This is especially significant since vector architectures are supposed to be relatively latency tolerant. This data shows that for traditional vector machines, this is not the case.

If we look now at the effects of reducing the vector register length, we can see that performance degradation is very high. The conclusion is that, as expected, reducing the vector register length in a traditional vector machine results in a remarkable loss of performance. The cost savings are clearly out-weighted by the execution time degradation. Unless some latency tolerance technique is added to a traditional vector machine, vector register length should be kept as long as possible. In the next section we will see how decoupling can compensate this performance loss.

6 Combining short vectors and decoupling

In this section we will study how the combination of a latency tolerance technique such as decoupling can be combined with a vector architecture having short registers to overcome the performance degradation seen in the previous section. As we will see, decoupling with short registers can even provide speedups with respect to a traditional in-order machine.

6.1 Decoupled Vector Architecture

For ours simulations we used the decoupled vector architecture introduced in [9]. The main idea in this architecture is to use a fetch processor to split the incoming, non-decoupled, instruction stream into three different decoupled streams (see fig. 6). The translation is such that each processor can proceed independently and, yet, synchronizes through the communication queues when needed. Each of these three streams goes to a different processor: the address processor

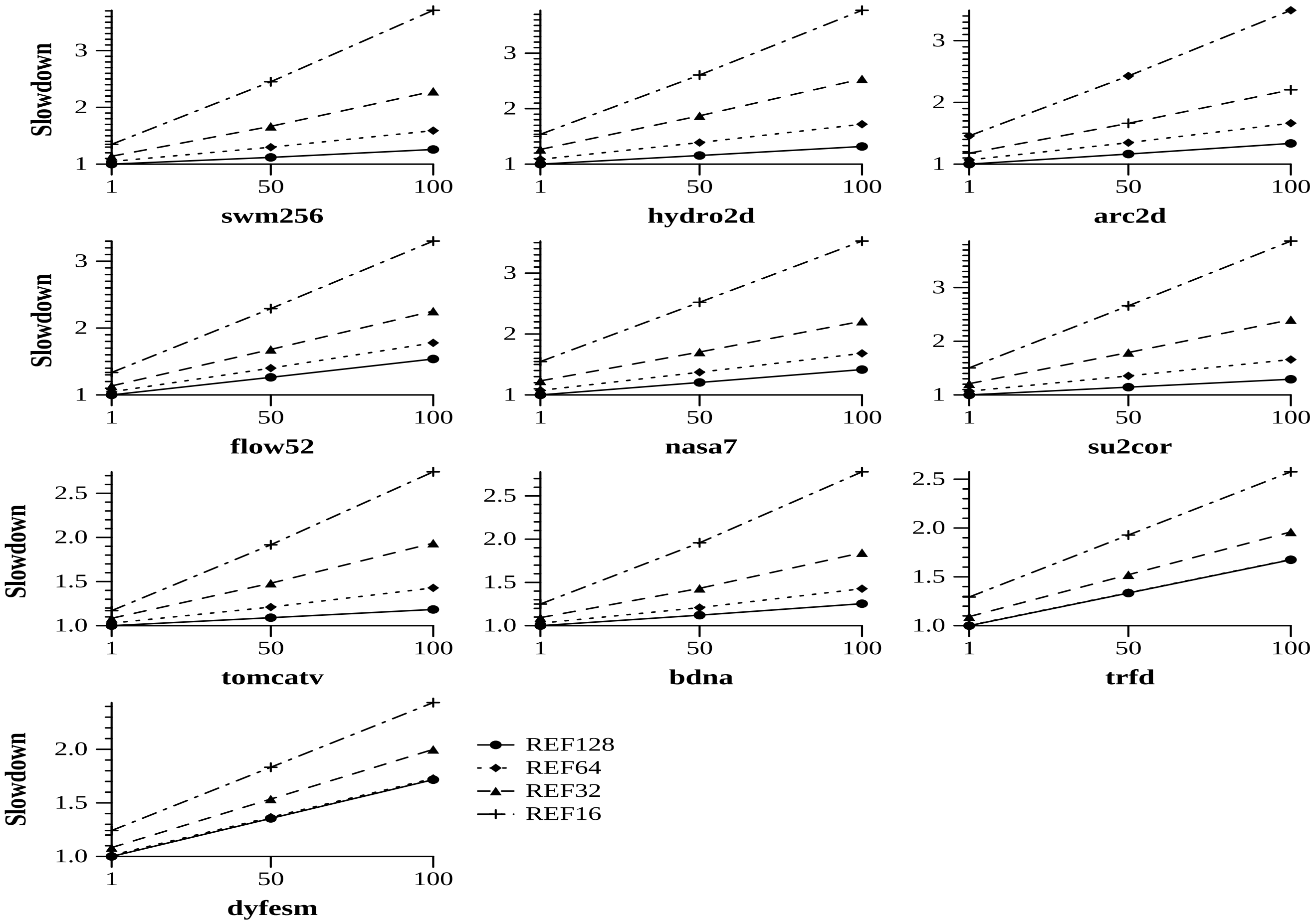

Figure 5: Effects of memory latency and vector register length on performance. X-axis is memory latency.

(*AP*), that performs all memory accesses on behalf of the other two processors, the scalar processor (*SP*), that performs all scalar computations and the vector processor (*VP*), that performs all vector computations. The three processors communicate through a set of *implementational* queues and proceed independently. This set of queues is akin to the implementational queues that can be found in the floating point part of the R8000 microprocessor[13]. The main difference of this decoupled architecture with previous scalar decoupled architectures such as the ZS-1 [14], the MAP-200 [15], PIPE [16] or FOM [17], is that it has *two* computational processors instead of just one. These two computation processors, the *SP* and the *VP*, have been split due to the very different nature of the operands on which they work (scalars and vectors, respectively).

The main parameters of this architecture are the length of its queues: the three instruction queues, the inter-processor queues, the scalar queues and the load store address queues were set at 16 elements. For the vector queues (numbers 1 and 2), each slot is a full vector register and, therefore, their size has to be carefully considered. We start with 4 slots in each of them, as suggested in [9]. Reducing the vector register length benefits a decoupled implementation since each slot in the extra queues required to decouple the machine can be smaller than in the original machine.

The key points in this architecture will be to achieve good performance with relatively few slots in these two queues. This is another point where reducing the vector register length can be very helpful.

6.2 Performance of the DVA

What is the performance of the decoupled machine using different vector register lengths ? Figure 7 plots the simulated performance for the decoupled and non-decoupled machines for several memory latencies. For each program, we plot the baseline performance of the non-decoupled machine with a register length of 128 and the performance of the decoupled versions using register lengths of 16, 32, 64 and 128. Note that the Y-axis plots the relative performance of each configuration relative to the non-decoupled machine with length 128 and memory latency of 1 cycle. Thus, in figure 7 numbers above 1.0 indicate a *slowdown* and numbers below 1.0 indicate *speedups*.

We will start comparing the performance of the decoupled and non-decoupled machines with the maximum vector register length (128). As already presented in [9], the performance improvements due to decoupling are quite substantial. Even with a perfect

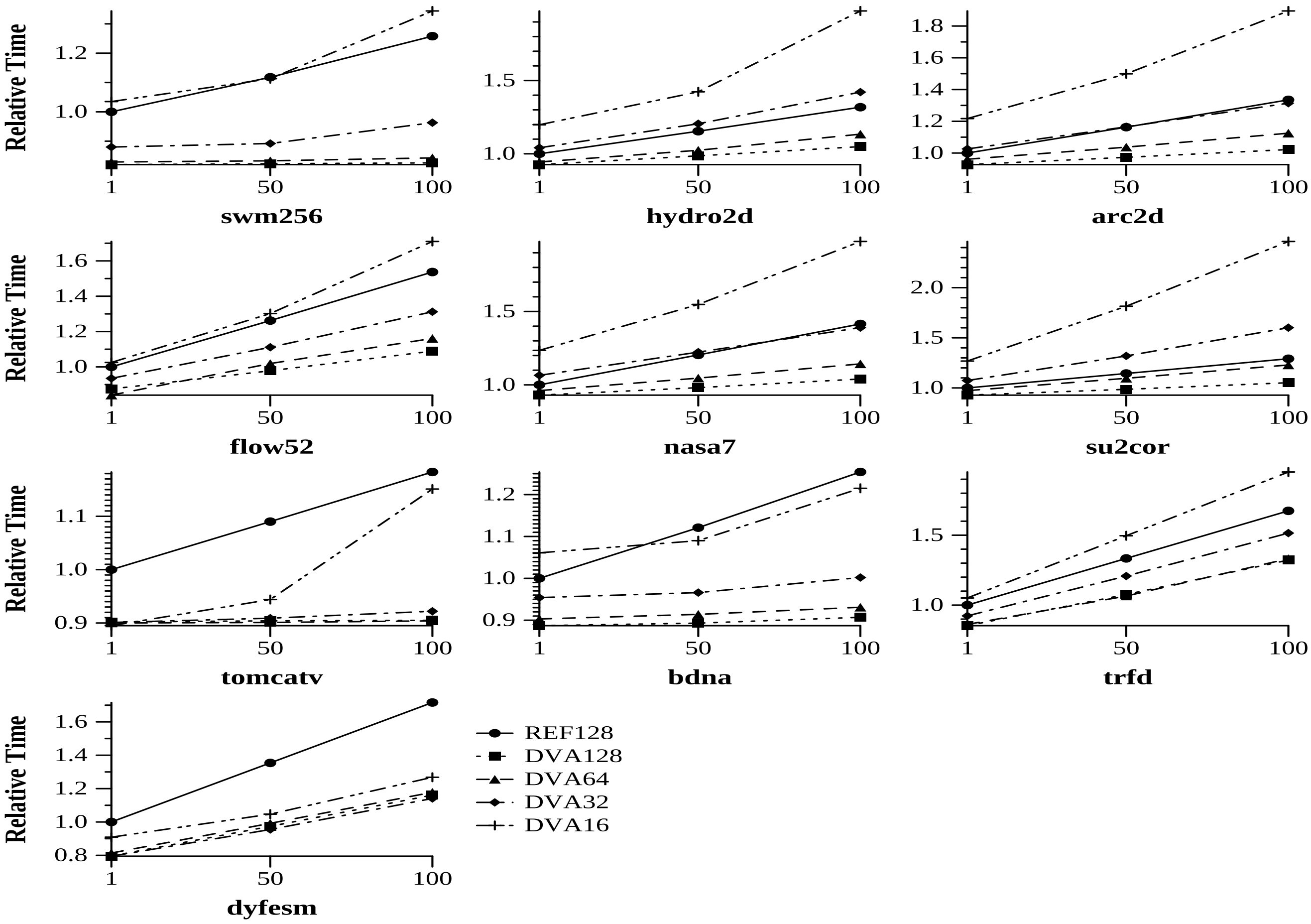

Figure 7: Effects of memory latency and vector register length on performance when using decoupling.

memory system with latency 1, speedups are in the range 1.10–1.25. When memory latency is increased up to 100 cycles, the DVA experiences some slowdowns, but much smaller than the reference machine. Comparing both machines at a latency of 100, the DVA yields speedups in the 1.22–1.52 range.

When the register length is reduced we still obtain very good results. Halving the register length (64 elements), yields a machine that performs only worse than the DVA128 by factors of 1.01–1.10 but that, in *all* cases performs much better than the reference machine. Comparing performance at 100 cycles memory latency, we see speedups of the DVA64 over the REF machine in the 1.05–1.49 range. Note that, in three cases, the performance of the DVA64 at 100 cycles latency is *better* than the REF machine performance at 1 cycle memory latency. In all programs but `trfd` and `su2cor`, if we compare the DVA64 at 100 cycles and the reference machine at 50 cycles we see that the decoupled machine performs better (by factors in the range 1.01–1.32). These results suggest that even halving the register length, a machine with a slower memory system (thus, a much cheaper memory system) would perform better than a traditional machine.

Reducing the register length to 1/4 of the original length (32 elements), we still see that the performance of the DVA32 is better than the reference machine. Except for programs `hydro2d`, `nasa7` and `su2cor`, the DVA32 achieves speedups over the REF machine in the range 1.01–1.25 and goes up to 1.42 for `dyfesm` (at latency 50).

Only when the register length is reduced to 16 elements (1/8 of the original) performance starts to degradate noticeably. Seven out of ten programs perform worse with the DVA16 than with the REF machine, and only `dyfesm` and `tomcatv` maintain a good performance. This sudden jump in execution time is due to the combination of several effects: the number of scatter/gather operations, the number of outstanding branches and dependencies in scalar code introduce many cycles of stall in a program run. These three types of hazards stall the vector processor very frequently, thereby exposing the full memory latency at each memory load being executed. This explains the steep slopes of each of the DVA16 curves.

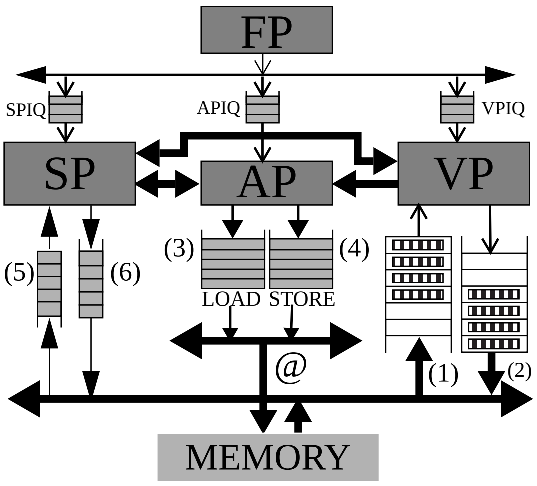

Figure 6: The decoupled vector architecture studied in this paper. Queue names: (1) vector load data queue –VLDQ, (2) vector store data queue –VSDQ, (3) address load queue –ALQ, (4) address store queue –ASQ, (5) scalar load data queue –SLDQ, (6) scalar store data queue –SSDQ,

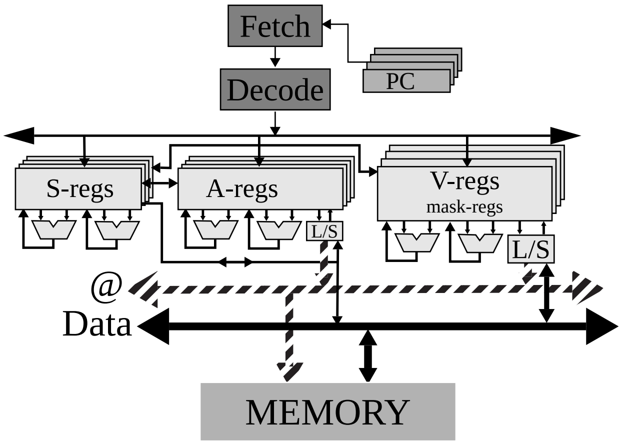

Figure 8: The Multithreaded vector architecture studied in this paper.

7 Combining short vectors and Multithreading

This section will evaluate the benefits of adding multithreading to a traditional in-order machine with short registers. While the DVA machine presented in the previous section focused on improving single thread performance, multithreading is targeted at improving overall throughput. In both cases, the vector register reduction can be an advantage if combined with a latency tolerance technique. In a multithreaded machine, reducing the vector length to 1/2 of the original size can allow running two independent threads on (almost) the same hardware as the original machine.

7.1 Multithreaded Vector Architecture

The multithreaded vector architecture (MVA) we propose is also modeled after a Convex C3400 architecture and was introduced in [10]. The multithreaded version of the reference architecture is shown in figure 8. 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. Nonetheless, multithreading does not come without problems. The cost of the read/write crossbars between the registers and the functional units is a limiting factor to be considered. 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. To take this into account, we charge the multithreaded processor with an extra cycle both for reading and writing from/into the vector register file.

7.2 Simulation methodology

To evaluate the performance of the multithreaded processor under different memory latencies, we have to change the benchmarking strategy. We need to fix the total amount of work being simulated. We will define a new benchmark consisting in the execution of ALL 10 programs used so far. These 10 programs are ordered randomly (in particular, the order chosen is `flo52 (tf)`, `swm256 (sw)`, `su2cor (su)`, `trfd (ti)`, `tomcatv (to)`, `nasa7 (a7)`, `hydro2d (hy)`, `bdna (na)`, `arc2d (sr)`, `dyfesm (sd)`). Then, when

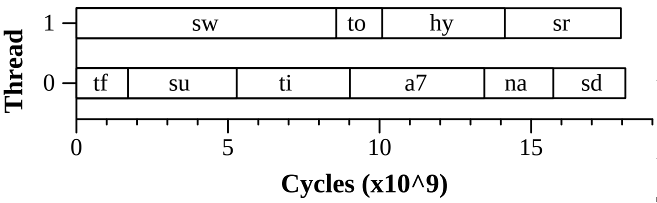

Figure 9: Execution example of the 10 programs run on a 2-context machine with a memory latency value of 50 cycles.

doing simulations of a, say, 4 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 [18], 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.

7.3 Performance of the MVA

The design space resulting from the combination of short registers and multithreading is relatively larger than what we saw for the decoupled machine. Given a certain budget in terms of register storage space, the processor designer can trade-off the number of threads against the length of each individual register. For example, consider the baseline machine (REF128). This machine has a vector register file that can hold $8 \times 128 \times 8 = 8Kb$ of storage. When multithreading the machine, the designer can choose between several alternatives: halving each register down to 64 elements and allowing 2 hardware contexts or reducing each register to 32 elements and allowing 4 hardware contexts, etc. On the other hand, a designer could decide to lower the overall cost of the machine by only implementing a 4Kb register file and splitting it between several threads, hoping that the instruction level parallelism and latency tolerance introduced by the multithreading technique would compensate both the reduction in total storage and the reduction in each individual register as seen by a single thread.

We start by exploring different configurations assuming that the number of threads is 2. Later we will expand this number to 4 threads. Figure 10 presents the execution time for the sequence of ten benchmarks on several multithreaded machines. Assuming only two threads, the register storage space can be partitioned in several ways: 2 threads with each register 64 elements long (curve `2@64`), 2 threads with each register being 32 elements long (curve `2@32`) or 2 threads with each register being 16 elements long (curve `2@16`). Note that these three architectures have, respectively, 8Kb, 4Kb and 2Kb of total storage

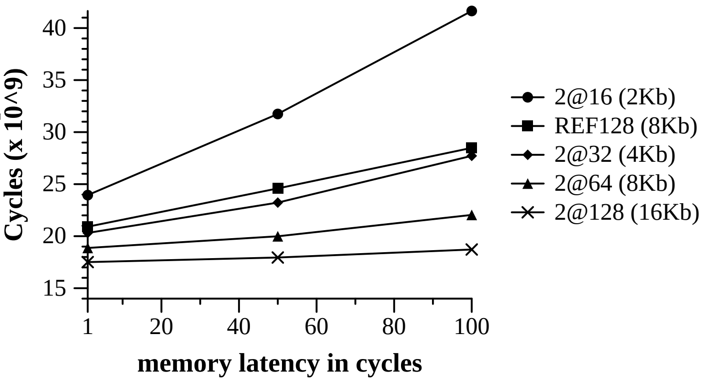

Figure 10: Effects of memory latency and vector register length on performance in the multithreaded architecture.

space in their vector register file. For comparison, we also include a multithreaded machine having 16Kb of storage (2 threads @ 128 elements, curve `2@128`) and the reference machine (8Kb of space, curve `REF128`).

Figure 10 shows a pattern similar to the behavior of the decoupled architectures. If we start looking at the benefits of fully multithreading the reference machine using 2 contexts with the original registers of 128 elements (curve `2@128`), we can see that the benefits of multithreading are rather large. At latency 1, the speedup over the REF128 machine is 1.19 and goes up to 1.52 at latency 100 cycles. If we start trading off register length for hardware contexts, we can see that introducing two threads at the expense of halving each register (curve `2@64`) is still a good decision performance wise. The `2@64` machine uses the same storage space and a slightly more complex crossbar than the REF128 machine but, despite the extra cycle charged to traverse each crossbar, it yields speedups in the range 1.10–1.29. Reducing register size to 32 and keeping only 2 threads (curve `2@32`) incurs a severe degradation but still outperforms the REF128 machine. The `2@32` configuration could be a good tradeoff to reduce total machine cost while retaining a similar level of performance. Finally, going down to the `2@16` machines where each register is only 16 elements long is much worse than using a traditional machine, something similar as to what happened in the decoupled architectures.

Can performance be improved by using more threads where possible ? For example, if our budget is 8Kb of registers, is it better to have 2 threads at 64 elements or 4 threads at 32 elements ? Figure 11 answers these questions. This figure plots, for three different register capacities (4, 8 and 16Kb), the execution time for configurations that split the available capacity in symmetric ways. Due to space constraints, we only present results for a memory system having 50 cycles latency. Figure 11 shows that the difference between 2 and 4 threads is very small if the total register space is kept constant. That is, in general, it is better to have fewer threads with longer vector registers than more threads with shorter vector registers. This

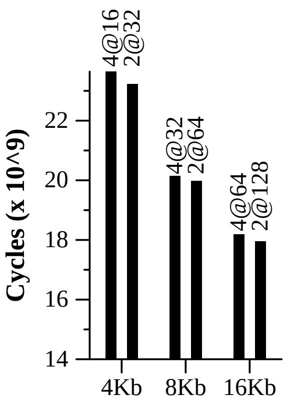

Figure 11: Execution time comparison of 2 versus 4 threads when given a certain vector register budget.

conclusion, though, deserves more investigation with programs that are either (a) less vectorizable or (b) have smaller vector lengths than our ten benchmarks, before being settled.

8 Summary

This paper has presented data on the tradeoffs involved in choosing an adequate vector register size for vector ISAs. Traditionally, very large vector registers have been chosen to maximize the amount of latency amortized per vector instruction. Nonetheless, this election was made in an environment where almost all vector architectures executed instructions in strict program order (with some minor overlapping between vector and scalar instructions). Despite the need for very long registers, many highly vectorizable programs can not make full use of every single element in a register. Our measurements show how in many programs, less than 40% of all register being used are completely filled with 128 elements of data. Unfortunately, our simulations confirm that it is not possible to reduce the vector register length in a traditional vector architecture without severely affecting performance: halving the register length, for example, yields slowdowns in the range 1.05–1.8.

Nonetheless, since the first register-to-register vector ISA were introduced, the superscalar community has shown that many other techniques can be used to tolerate long latencies, such as the typical long delays associated with memory accesses. These techniques are based on exploiting instruction level parallelism (ILP) by executing instructions out-of-order, in a decoupled way or by using multithreading.

This paper has shown that when ILP is exploited using either decoupling or multithreading the need for very large vector registers is substantially reduced. The reduction in vector register length can be used in two different ways: either to decrease processor cost by reducing the total amount of storage devoted to register values or to improve performance by more effectively using the available storage by sharing it between several threads or by adding vector queues in a decoupled environment.

Simulations show that combining decoupling and short registers it is possible to reduce the size of each vector register to 1/2 with a good performance improvement (speedups of 1.05–1.49) and down to 1/4 at a similar level of performance (speedups of 1.01–1.25) although some programs might experience small slowdowns (less than 5%). The overall register space requirements for the DVA32 machine is half the original reference machine.

Simulations also show that combining multithreading and short vectors yields substantial speedups. Two threads can be accommodated in the original register space providing speedups in the range 1.23–1.29. Further reducing the register size to 32 elements is only marginally better than the reference machine (1.02 to 1.05 speedups) but halves the register file cost.

We believe that the results presented in this paper are not only relevant to the vector processor community but could also be of use in the near term for designers of multimedia instruction sets [19, 20, 21] [22, 23]. A multimedia instruction set operates on short vectors (typically, no more than 8 elements) by exploiting sub-word parallelism. Our results show that increasing the multimedia ISA's to accommodate vectors of 16 or 32 elements might provide significant speedups.

References

[1] R. G. Hintz and D. P. Tate. Control data STAR-100 processor design. In *Proc. Compcon 72*, pages 1–4, New York, 1972. IEEE Computer Society Conf. 1972, IEEE.

[2] W. Watson. The TI-ASC, A highly modular and flexible super computer architecture. *Proc. AFIPS*, 41, pt. 1:221–228, 1972.

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

[4] W. Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18(8):947–954, August 1992.

[5] Katsuyoshi Kitai, Tadaaki Isobe, Tadayuki Sakakibara, Shigeko Yazawa, Yoshiko Tamaki, Teruo, and Kouichi Ishii. Distributed storage control unit for the Hitachi S-3800 multivector supercomputer. In *ICS*, pages 1–10, July 1994.

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

[7] Robert Perron and Craig Mundie. The architecture of the Alliant FX/8 computer. In *Digest of Papers: Compcon 86*, pages 390–394, Washington, D.C., March 1986.

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

[10] Roger Espasa and Mateo Valero. Multithreaded vector architectures. In *HPCA-3*, pages 237–249. IEEE Computer Society Press, Feb 1997.

[11] Roger Espasa, Mateo Valero, and James E. Smith. Out-of-order Vector Architectures. Technical Report UPC-DAC-1996-52, Univ. Politècnica de Catalunya–Barcelona, November 1996.

[12] R. A. Fatoohi. Vector performance analysis of three supercomputers: Cray 2, cray Y-MP, and ETA 10-Q. In *Proceedings of the Supercomputing 89*, pages 779–788, Reno, NV USA, 1989. ACM Press , New York, NY , USA.

[13] P.Y.T Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[14] James E. Smith, G.E. Dermer, B.D. Vanderwarn, S.D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon. The ZS-1 Central Processor. In *ASPLOS-II*, pages 199–204. CS press, 1987.

[15] E. U. Cohler and J. E. Storer. Functionally parallel architectures for array processors. *Computer*, 14:28–36, September 1981.

[16] J.R. Goodman, J.T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young. PIPE: A VLSI Decoupled Architecture. In *ISCA*, pages 20–27, June 1985.

[17] W. C. Brantley and Joseph Weiss. Organization and architecture tradeoffs in FOM. In *IEEE International Workshop on Computer Systems Organization*, March 1983.

[18] 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.

[19] Craig Hansen. MicroUnity's MediaProcessor Architecture. *IEEE Micro*, pages 34–41, August 1996.

[20] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, pages 10–20, August 1996.

[21] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, pages 42–50, August 1996.

[22] L. T. Nguyen, M. Mohamed, H. Park, Y. Pai, R. Wong, A. Qureshi, P. Psong, F. Valesco, H. D. Truong, and C. Reader. Multi-media Signal Processor (MSP) summary. In *Hot Chips VIII*, pages 217–226, Palo Alto, CA, August 1996. Samsung Semiconductor.

[23] Krste Asanovic, James Beck, Bertrand Irissou, Brian Kingsbury, Nelson Morgan, and John Wawrzynek. The T0 Vector Microprocessor. In *Hot Chips VII*, pages 187–196, August 1995.