Effective Usage of Vector Registers in Decoupled Vector Architectures

Luis Villa * Roger Espasa† Mateo Valero

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

Abstract

This paper presents a study of the impact of reducing the vector register size in a decoupled vector architecture. In traditional in-order vector architectures, long vector registers have typically been the norm. We start presenting data that shows that, even for highly vectorizable codes, only a small fraction of all elements of a long vector register are actually used. We also show that reducing the register size in a traditional vector architecture in an attempt to reduce hardware cost and maximize register utilization results in a severe performance degradation. However, we combine the decoupling technique with the vector register reduction and show that the resulting architecture tolerates very well the register size cuts. We simulate a selection of Perfect Club and Specfp92 programs using a trace driven approach and compare the execution time in a conventional vector architecture with a decoupled vector architecture using different registers sizes. Halving the register size and using decoupling provides speedups between 1.04–1.49 over a traditional in-order vector machines. Even reducing the register length to 1/4 the original size (and, in some cases, to 1/8) the performance of the decoupled machine is better than a conventional vector model. Moreover, we observe that the resulting decoupled machine with short registers tolerates very well long memory latencies.

1 Introduction

Vector architectures have been used for many years for high performance numerical applications – an area where they still excel.

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.

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, then reducing the vector register length will reduce cost and increase the fraction of usage of

*On leave from the Centro de Investigación en Cómputo, Instituto Politécnico Nacional – México D.F. This work was supported by the Instituto de Cooperacion Iberoamericana (ICI), Consejo Nacional de Ciencia y Tecnologia (CONACYT).

†This work was supported by the Ministry of Education of Spain under contract 0429/95, and by the CEPBA.

0-8186-8332-5/98 $10.00 © 1998 IEEE

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 [1, 2, 3].

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 an ILP technique, decoupling, the performance penalty can be made very small. We will show that 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 architecture 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 usage

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. In [4] we evaluated a set of highly vectorizable applications in order to know which was the vector length used by these programs.

The first thing to note is that, even though these set of 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.

Which will be the effective register length use if we vary the vector register size ?? In [4] answered this question showing how the application vector length and the hardware vector register length are related. We noted 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.

```
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(32)
   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 1: (a) `flo52` loop without Strip-Mining, (b) Adding Strip-mining.

3 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 [5] 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 1). 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 [5]. This directive informed the compiler that the inner loop was performing less than 128 trips and thus no extra strip-mining was generated.

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 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%.

4 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.

4.1 Performance on the Reference Architecture

Our reference machine is loosely based on a Convex C3400. The essential characteristics of the reference architecture are a single memory port, two functional units and 8 vector register. In [6] we give a detailed explanation of this reference architecture. In [4] we studied four different variants of this reference machine. The four models under study was referred to as the REF128, REF64, REF32 and REF16 architectures with a vector length of 128, 64, 32 and 16 elements respectively.

We noted that the impact of memory latency is very significant. For our unmodified model (REF128) we observed that execution time is degraded by factors of 1.2–1.4 in most programs when we vary the latency from 1 to 100 cycles.

We observed that reducing the vector register length performance degradation is very high. Our conclusion were that, reducing the vector register length in a traditional vector machine results in a remarkable loss of performance. The cost savings are clearly outweighted 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.

5 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.

5.1 Decoupled Vector Architecture

For ours simulations we used the decoupled vector architecture introduced in [1]. 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. 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 (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[7]. The main difference of this decoupled architecture with previous scalar decoupled architectures such as the ZS-1 [8] or the MAP-200 [9] 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 [1]. 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.

5.2 Performance of the DVA

What is the performance of the decoupled machine using different vector register lengths ? Figure 2 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 2 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 [1], the performance improvements due to decoupling are quite substantial. Even with a perfect 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

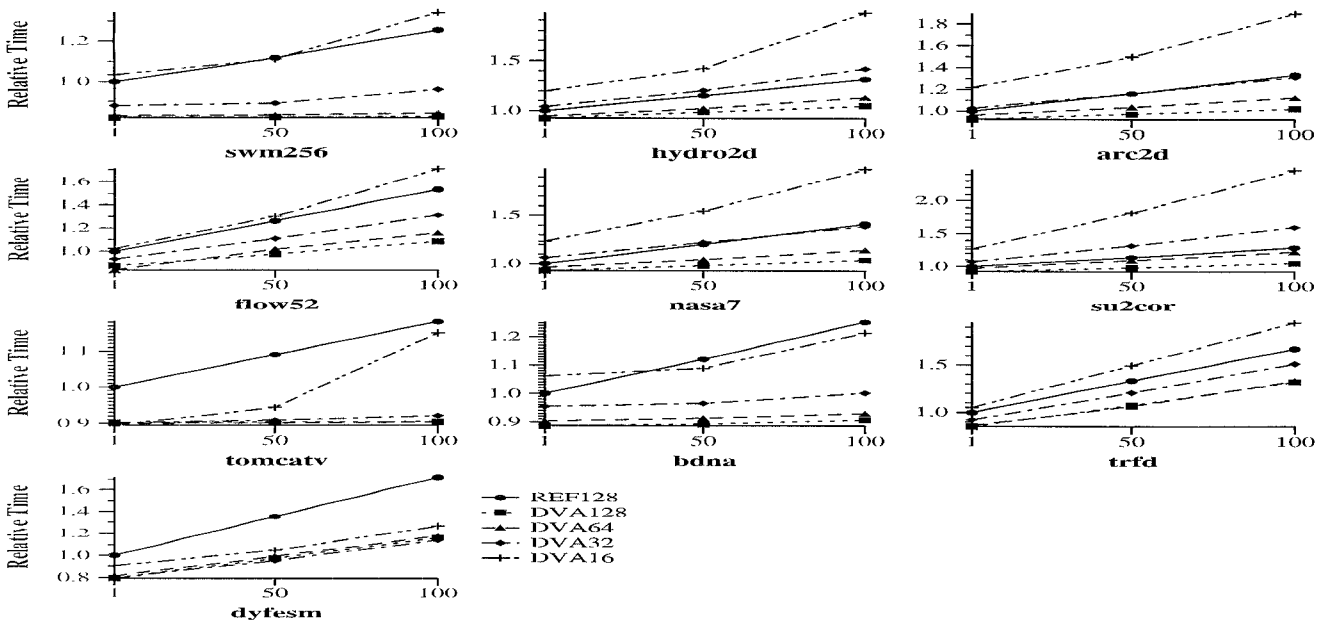

Figure 2: Effects of memory latency and vector register length on performance when using decoupling.

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.

6 Increasing Queue Length

The load and store queue length is a key parameter in a decoupled architecture. It determines the amount of data that can be prefetched ahead of time and, therefore, the queue length puts an upper limit on the maximum memory latency that can be tolerated. For example, a system having 8 slots in the load queue, each corresponding to a 32 element vector can request up to $8 \times 32 = 256$ data items to the memory system before blocking. If main memory latency is shorter than 256 cycles, then this decoupled system can establish a continuous flow of data from main memory into the processor without stalls (provided there are enough load instructions to keep the pipeline fed, of course). On the other hand, if memory latency (L) is larger than 256 cycles, no matter how fast we can feed the address processor, the flow of requests to the memory system will be interrupted and a fraction of all memory latency ($L - 256$) will be exposed to the computation processor.

In this section we will look at the performance improvement due to enlarging both the load and store queue lengths. We expected that, the longer the queue, the better memory latency will be tolerated. As we will see, this intuition is wrong and there is a limit after which increasing queue length does not yield any significant performance advantage.

Figure 3 presents for our ten benchmark programs the improvements due to increasing the queue size. Due to lack of space we present only the data for the DVA16 model, where each vector register is supposed

to hold 16 elements. For each program we plot 4 different bars, labeled "Q=4" through "Q=32" that indicate the number of slots in the vector load queue and the vector store queue. For these 4 bars, memory latency was assumed to be 50 cycles. Moreover, at the top of each bar there is a white bar representing the execution time for the same queue size but assuming a memory latency of 100 cycles.

As it can be seen from this figure, increasing from 4 slots to 8 slots does provide some performance improvement, especially at 100 cycles memory latency, but further increasing the queue to 16 or 32 slots does not provide any additional benefits. The result is striking if we compare the total area requirements of the DVA128 and DVA16 architectures. For example, the DVA128 architecture holds a total of 128×4 items in each of its load and store queues. Similarly, the DVA16 architecture with 32 slots in the queues can potentially hold exactly the same amount of data (16×32), and yet it achieves a much worse performance. Although not presented here, a similar effect happens with the DVA32 and DVA64 architectures.

The overall conclusion is that increasing queue size does not compensate for the reduction in vector register length. This will be further analyzed in the following section.

7 Limits on performance

In what ways does reducing the vector register length limit performance ? As we have seen in the previous section, vector length reduction can no be compensated increasing the depth of the queues or trying to augment the ILP inside the computation processor. This section will analyze the causes of this behavior.

Latency Masking

The most important effect of reducing the vector length is that many latencies that were previously hidden underneath the execution of vector code are now exposed in the critical path of the program. This effect is represented in figure 4. On the left, we present a fraction of code from the most important loop of program `su2or`. This loop presents a true dependency from instruction 3 into instruction 4. Let's assume that the latency for performing an addition on our architecture is 3 cycles. The schematic code sequence shown in (a) displays the behavior of this loop under the DVA128 model. Each vector instruction performs 128 operations and there is a 3 cycle stall between the start of instruction 3 and instruction 4. We note that this 3 cycle stall is in the critical path of the loop. Under this model, each iteration of the loop would take $128 \times 3 + 2 + 3 = 389$ cycles, assuming that the two scalar operations are executed in a single cycle each. Therefore, the percentage of wasted cycles (3/389) is very small (0.77%). Executing the same loop under the DVA64 model (example shown in (b)), we will still pay the same 3 cycles of stall, but they will be amortized over less elements. The percentage

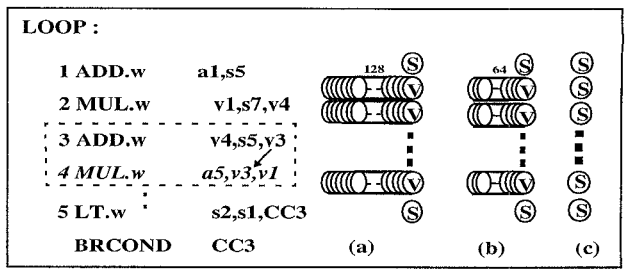

Figure 4: Effects of dependences for different vector register lengths.

of wasted cycles will be $3/(64 \times 3 + 2 + 3) = 1.5\%$. Finally, in the extreme case, corresponding to a scalar machine (vector length = 1) shown in (c), we would pay 3 stall cycles every 8-cycle iteration, yielding a waste of 37.5%.

Another way of looking at figure 4 is to consider that the 3 cycles of stall involved in the dependency will have to be payed for each data item processed in loop (c), for every 64 data items processed in loop (b) or for every 128 items processed in loop (a). Thus, in order to execute a given amount of work architecture (a) will take less time than architecture (c).

The overall lesson is that the more the vector register length is reduced, the more this small latencies are exposed to total execution time. In a vector architecture having 128 elements per register, a 3 cycle latency is almost hidden, whereas on a scalar machine this latency is exposed on every single iteration.

Note that we are not claiming here that the scalar execution model is necessarily worse than the vector model. There are many techniques (loop unrolling, software pipelining, etc.) that could help improve the performance of the loop as executed on (c). We are simply pointing out that, given the binaries as they are, a decrease in vector length will expose more latencies (both from main memory and from functional units) and will increase a program's critical path. The increase in total execution time is proportional to the decrease in vector register length.

Gather-Scatter instructions

Another very important limitation to performance in a decoupled vector architecture is the amount of gather/scatter instructions in the code. A gather instruction can not be characterized with a memory range, and thus imposes a sequential bottleneck in the otherwise out-of-order execution of load/store instructions. Moreover, a gather instruction requires a vector from the *VP* before being able to proceed to the memory system. Thus, each time a gather has to be executed, a loss of decoupling appears: the *VP* and the *AP* have to synchronize to launch the gather instruction. No matter how much ahead the *AP* was from the *VP* it will have to wait until the *VP* provides the vector register with the required addresses.

Figure 5 shows a typical example of gather/scatter code from program `su2cor`. The `gather` instruc-

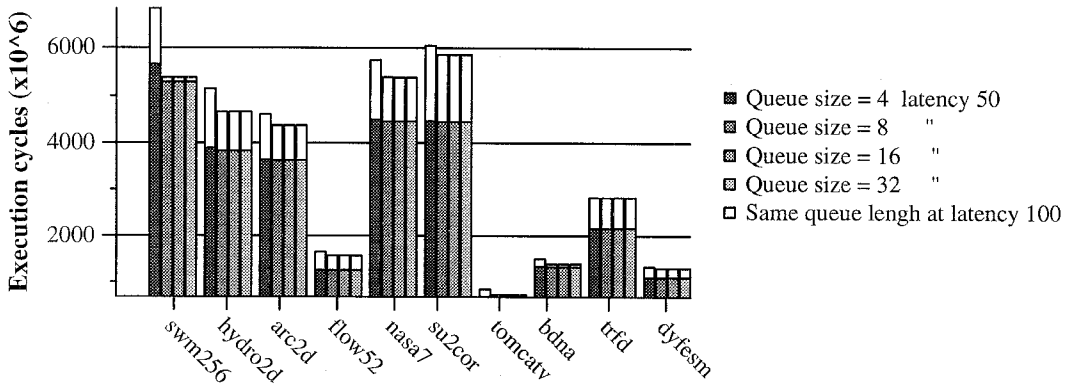

Figure 3: Performance of the DVA16 architecture for different queue sizes (4,8,16,32) and two memory latencies (50 and 100 cycles).

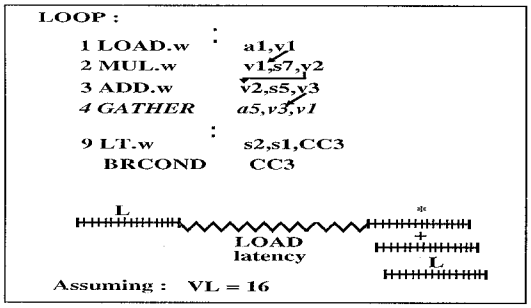

Figure 5: Structure of gather-scatter code.

Programs	128	64	Ratio
SWM256	3.7	3.7	1.0
HYDRO2D	0.7	11.8	16.8
ARC2D	2.0	4.3	2.2
FLO52	2.4	5.8	2.4
NASA7	7.4	10.8	1.5
SU2COR	15.5	19.5	1.3
TOMCATV	0.7	1.4	2.0
BDNA	6.0	6.0	1.0
TRFD	0.9	18.9	21.0
DYFESM	10.2	22.6	2.2

Table 1: Absolute number of mispredictions (in millions) for the 128 to 16 architectures

tion requires the computations carried out by instructions 2 and 3 before being able to proceed. Unfortunately, instruction 2 requires a register (`v1`) which must be loaded from memory. The time diagram in the lower part of figure 5 shows the latency exposure introduced by the gather instruction. A full memory latency plus the latency of an `add` and a `mul` operation must elapse before the `gather` instruction can proceed. This full memory latency can not be used to dispatch other loads because the decoupled architecture executes loads in-order. It can not be used to dispatch younger stores precisely because a gather instruction can not be characterized with a memory range and thus, the hardware must conservatively assume a dependency between a gather and all following store instructions.

As already mentioned in the previous case, the number of times that this full memory latency is exposed is proportional to the length of the vector registers. In the DVA16 model, this memory latency will be exposed 8 times more than in the DVA128 model, thus partially contributing to the slowdowns of the DVA16 machine. The longer the memory latency, the worse is this effect in the DVA16 case.

Branch Penalties

Another effect of reducing the vector register length is the increase of mispredicted branches. Table 1 presents the total number of mispredicted branches for each program, for the DVA128 and the DVA16. Note that the table presents *absolute number* of mispredictions rather than misprediction rate because the number of branches in each architecture varies (in fact misprediction rate is higher for the DVA128 machine because it executes much fewer branches overall).

As it can be seen from table 1, the number of mispredictions can greatly increase. For most programs, this effect is due to the following. The number of branch instructions in the program under either model is essentially the same. The effect of reducing vector register length is that each branch is visited more times. Those branches that were difficult to predict or that had conflicts with other branches and resulted in misses in the BTB in the DVA128 model, are executed many more times under the DVA16 model. Thus, total number of misprediction increases.

This explanation is clearly not enough in the case of `trfd` and `hydro2d`, which have an increase of mispredictions of 21.0 and 16.8 respectively. We investigated

the two programs and found that the increase was due to a combination of our strip-mining and short vector trips. The real vector length register in the C3 machine works in such a way that, if a value larger than 128 is written to it, it is automatically chopped down to 128. The compiler relies on this hardware behavior to save one test in its strip-mined code. By contrast, our manual strip-mining, although achieved the desired effect of emulating a machine with a smaller hardware vector length, can not rely on this effect and requires an extra comparison and jump to implement a `MIN(16,J)` operation. In `trfd`, the variable `J` takes values from 10 to 40 in steps of 5, causing the two-bit saturating counter to mispredict the jump most of the time.

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.

This paper has shown that when ILP is exploited using decoupling the negative impact of reducing the register length 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 adding vector queues in a decoupled environment. The overall effect is that very large registers in the decoupled context are no longer needed.

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 non-decoupled reference machine.

We have seen that there is a limit to the maximum possible reduction of the vector register length. Due to the increase of mispredicted branches, and the scheduling limitation imposed by Gather/Scatter operations, if the register length is reduced down to 16 elements, many stall cycles appear in the critical path of a program. Moreover, our simulations have shown that it is not possible to overcome these effects by enlarging the vector queues. Nonetheless, we are currently working in using the dynamic load/store elimination techniques described in [1] in our decoupled machine with short registers. The results show that in many cases, if bypassing is allowed between the store and the load queue, the performance of the DVA16 machine can be greatly improved.

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 [10] [11].

References

[1] Roger Espasa and Mateo Valero. Decoupled vector architectures. In *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, pages 281–290. IEEE Computer Society Press, Feb 1996.

[2] Roger Espasa and Mateo Valero. Multithreaded vector architectures. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 237–249. IEEE Computer Society Press, Feb 1997.

[3] Roger Espasa, Mateo Valero, and James E. Smith. Out-of-order Vector Architectures. In *MICRO-30*. IEEE Press, 1997.

[4] Luis Villa, Roger Espasa, and Mateo Valero. Effective usage of vector registers in advanced vector architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT97)*, San Francisco Cal., 1997.

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

[6] 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.

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

[8] James E. Smith, G.E. Dermer, B.D. Vanderwarn, S.D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon. The ZS-1 Central Processor. In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–204. CS press, 1987.

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

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

[11] 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.