Decoupled Vector Architectures

Roger Espasa Mateo Valero

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

Abstract

The purpose of this paper is to show that using decoupling techniques in a vector processor, the performance of vector programs can be greatly improved. Using a trace driven approach, we simulate a selection of the Perfect Club programs and compare their execution time on a conventional vector architecture and on a decoupled vector architecture. Decoupling provides a performance advantage of more than a factor of two for realistic memory latencies, and even with an ideal memory system with no latency, there is still a speedup of as much as 50%. A bypassing technique between the load/store queues is introduced and we show how it can give up to an extra speedup of 22% while also reducing total memory traffic by an average of 20%. An important part of this paper is devoted to study the tradeoffs involved in choosing an adequate size for the different queues of the architecture, so that the hardware cost of the queues can be minimized while still retaining most of the performance advantages of decoupling.

1 Introduction

Recent years have witnessed an increasing gap between processor speed and memory speed, which is due to two main reasons. First, technological improvements in cpu speed have not been matched by similar improvements in memory chips. Second, the instruction level parallelism available in recent processor has increased. Since several instructions are being issued at the same processor cycle, the total amount of data requested per cycle to the memory system is much higher. These two factors have led to a situation where memory chips are on the order of 10 to a 100 times slower than cpus and where the total execution time of a program can be greatly dominated by average memory access time.

This work was supported by the Ministry of Education of Spain under contracts TIC 880/92 and 0429/95, by ESPRIT 6634 Basic Research Action (APPARC) and by the CEPBA (European Center for Parallelism of Barcelona).

Current superscalar processors have been attacking the memory latency problem through basically three main types of techniques: caching, multithreading and decoupling (which, sometimes, may appear together). Cache-based superscalar processors reduce the average memory access time by placing the working set of a program in a faster level in the memory hierarchy. Software and hardware techniques such as [2, 15] have been devised to *prefetch* data from high levels in the memory hierarchy to lower levels (closer to the cpu) before the data is actually needed. On top of that, program transformations such as loop blocking [10] have proven very useful to fit the working set of a program into the cache.

Multithreaded processors [1, 22] attack the memory latency problem by switching between threads of computations so that the amount of parallelism exploitable augments, the probability of halting the cpu due to a hazard decreases, the occupation of the functional units increases and the total throughput of the system is improved. While each single thread still pays latency delays, the cpu is (presumably) never idle thanks to this mixing of different threads of computation.

Decoupled scalar processors [20, 17, 12] have focused on numerical computation and attack the memory latency problem by making the observation that the execution of a program can be split into two different tasks: moving data in and out of the processor and executing all arithmetic instructions that perform the program computations. A decoupled processor typically has two independent processors (the address processor and the computation processor) that perform these two tasks asynchronously and that communicate through architectural queues. Latency is hidden by the fact that usually the address processor is able to slip ahead of the computation processor and start loading data that will be needed soon by the computation processor. This excess data produced by the address processor is stored in the queues, and stays there until it is retrieved by the computation processor.

Vector machines have traditionally tackled the latency problem by the use of long vectors. Once a (memory) vector operation is started, it pays for some initial (potentially long) latency, but then it works on a long stream of elements and effectively amortizes this latency across all the elements. In vector multiprocessor systems the memory latency can be quite high due to conflicts in the memory modules and in the interconnection network. Although vector machines have been very successful during many years for certain types of numerical calculations, there is still much room for improvement. Several studies in recent years [16, 7] show how the performance achieved by vector architectures on real programs is far from the theoretical peak performance of the machine. In [7] is shown how the memory port of a single-port vector computer was heavily underutilized even for programs that were memory bound. It also shows how a vector processor could spend up to 50% of all its execution cycles waiting for data to come from memory.

Despite the need to improve the memory response time for vector architectures, it is not possible to apply some of the hardware and software techniques used by scalar processors because these techniques are either expensive or exhibit a poor performance in a vector context. For example, caches and software pipelining are two techniques that have been studied [11, 13, 21, 14] in the context of vector processors but that have not been proved useful enough to be in widespread use in current vector machines.

The conclusion is that in order to obtain full performance of a vector processor, some additional mechanism has to be used to reduce the memory delays (coming from lack of bandwidth and long latencies) experienced by programs. We will turn to the principle of decoupling as one of the techniques that can reduce the number of lost cycles due to memory problems.

The purpose of this paper is to show that using decoupling techniques in a vector processor, the performance of vector programs can be greatly improved. We will show how, even for an ideal memory system with no latency, decoupling provides a significant advantage over standard mode of operation. We will also present data showing that for more realistic latencies, decoupled vector architectures perform substantially better than non-decoupled vector architectures. Although in this paper we only look at the single processor case, the decoupling technique would also be very effective in vector multiprocessors to help reducing the negative effect of conflicts in the interconnection network and in the memory modules. We will also introduce a bypassing technique between the load/store queues and show how it can reduce the total execution time and also reduce the total memory traffic.

2 Experimental Framework

To asses the performance benefits of decoupled vector architectures we have taken a trace driven approach. The Perfect Club programs have been chosen as our benchmarks [8]. These programs are compiled on a Convex C3480 [3] machine and using Dixie [4] a detailed trace that describes its full execution is produced. The tracing procedure is as follows: the Perfect Club programs are compiled on a Convex C34 machine using the Fortran compiler (version 8.0) at optimization level -O2 (which implies scalar optimizations plus vectorization). Then the executables are processed using Dixie, a tool that decomposes executables into basic blocks and then instruments the basic blocks to produce four types of traces: a basic block trace, a trace of all values set into the *vector length* register, a trace of all values set into the *vector stride* register and a trace of all memory references (actually, a trace of the base address of all memory references). Dixie instruments all basic blocks in the program, including all library code. This is especially important since a number of fortran intrinsic routines (SIN, COS, EXP, etc.) are translated by the compiler into library calls. This library routines are highly vectorized and tuned to the underlying architecture and can represent a high fraction of all vector operations executed by the program. Thus it is essential to capture their behavior in order to accurately model the execution time of the programs.

Once the executables have been processed by Dixie, the modified executables are run on the Convex machine. This runs produce the desired set of traces that accurately represent the execution of the programs. This trace is then fed to two different simulators that we have developed: the first simulator is a model of the Convex C34 architecture and is representative of single memory port vector computers. The second simulator is an extension of the first, where we introduce decoupling. Using these two cycle-by-cycle simulators, we gather all the data necessary to discuss the performance benefits of decoupling.

2.1 The Reference Vector Architecture

We have designed a vector architecture, that we will refer to as the *Reference Vector Architecture* that is a close model of the C3400 architecture, albeit some of the low level details of the particular implementation of the C3400 have been overlooked. The main implication of this election is that this study is restricted to the class of vector computers having one memory port and two functional units. It is also important to point out that we used the output of the Convex compilers to evaluate our decoupled architecture. This means that the proposal studied in this paper is able to execute in a fully transparent manner an already existing instruction set.

The reference architecture consists of a scalar part and an independent vector part. The scalar part executes all instructions that involve scalar registers (A and S registers), and issues a maximum of one instruction per cycle. The vector part consists of two computation units (`FU1` and `FU2`) and one memory accessing unit (`LD`). The `FU2` unit is a general purpose arithmetic unit capable of executing all vector instructions. The `FU1` unit is a restricted functional unit that executes all vector instructions *except* multiplication, division and square root. Both functional units are fully pipelined. The vector unit has 8 vector registers which hold up to 128 elements of 64 bits each one. This eight vector registers are connected to the functional units through a restricted crossbar.

Program	#bbs	#insns S	V	#ops V	% Vect	avg. VL
ARC2D	5.2	63.3	42.9	4086.5	98.5	95
FLO52	5.7	37.7	22.8	1242.0	97.1	54
BDNA	47.0	23.9	19.6	1589.9	86.9	81
TRFD	44.8	352.2	49.5	1095.3	75.7	22
DYFESM	34.5	236.1	33.0	696.2	74.7	21
SPEC77	166.2	1147.8	262.6	3337.8	74.4	12
MG3D	452.14	11066.75	663.79	17995.49	61.9	27
MDG	185.90	4446.64	746.97	5611.66	55.8	7
ADM	42.4	709.0	104.3	694.8	49.5	6
OCEAN	165.64	4414.30	16.17	1444.92	24.7	89
QCD	80.05	1079.77	4.10	91.84	7.8	22
TRACK	50.67	505.96	4.47	26.96	5.1	6
SPICE	31.12	279.06	3.37	7.80	2.7	2

Table 1: Basic operation counts for the Perfect Club programs.

Every two vector registers are grouped in a register bank and share two read ports and one write port that links them to the functional units. The compiler is responsible to schedule the vector instructions and allocate the vector registers so that no port conflicts arise. The machine modeled chains vectors from functional units to other functional units and to the store unit. It does not chain memory loads to functional units, however. The real Convex C34 does not chain memory loads to functional units (nor do the Cray-2 and Cray-3). Although such chaining could be done, it is more complicated than other chaining because the memory system may not deliver the individual vector elements in order. We note that the Convex compiler used for our study schedules vector instructions taking the lack of load chaining into account. Because the modeled machine has two read pointers and one write pointer, all implemented chaining is fully flexible – chaining between two dependent instructions may be initiated regardless of the time the second issues.

2.2 The benchmark programs

Table 1 presents some basic facts about the thirteen Perfect Club programs. First column in this table presents the total number of basic blocks (in millions) executed for each program. Next two columns present the total number of instructions issued by the dispatch unit, broken down into scalar and vector instructions. Column four presents the number of operations performed by the vector instructions. Each vector instruction can perform several operations, hence the distinction between vector instructions and vector operations. Fifth column is the percentage of vectorization of each program. We define the degree of vectorization of a program as the ratio between the number of vector operations and the total number of operations performed by the program (i.e., column four divided by the sum of columns two and four). Finally column six presents the average vector length used by vector instructions, and is the ratio between vector operations and vector instructions (columns four and three, respectively).

One important point is that we want to evaluate the effects of decoupling for vector programs. Decoupling for scalar programs has already been studied in [19, 17, 18]. Our simulations have been biased towards a high level of detail in the vector portion of the architectures under study, and we have overlooked most of the details in the scalar portion of the cpu. Therefore, we require from the benchmark programs to be highly vectorizable ($\geq 70\%$) in order to render our results meaningful. We have selected six programs: ARC2D, FLO52, BDNA, SPEC77, TRFD and DYFESM.

3 Analysis of the Reference Architecture

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

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

Figure 1 presents the execution time of the six benchmark programs broken down into the eight possible states. For each program, we have plotted the execution time for four different values of memory latency. From this figure we can see that the fraction of cycles where these programs proceed at peak floating point speed (states $\langle FU2, FU1, LD\rangle$ and $\langle FU2, FU1,\ \rangle$) is not very high, and that it decreases as memory latency increases. Moreover, memory latency has a high impact on total execution time for programs DYFESM, TRFD and SPEC77, which have relatively small vector lengths. The effect of memory latency can be seen by noting the increase in cycles spent in state $\langle\ ,\ ,\ \rangle$. This increase can only be explained by the variation in memory latency. What is important to note, is that the sum of cycles corresponding to states where the LD unit is idle is quite high (51.9% for DYFESM, 48% for SPEC77, 35.1% for BDNA, 30.2% for TRFD, 11.13% for ARC2D and 10.58% for FLO52). These four states correspond to cycles where the memory port is idle and could (and should) be used to start fetching from memory the data that will be needed by the vector computations in the near future.

4 The Decoupled Vector Architecture

The decoupled vector architecture we propose splits the instruction stream into three different streams (see figure 2). One has the vector computation instructions only, and is executed by the vector processor (VP). The other contains all the memory accessing instructions (both vector and scalar) and goes to the address processor (AP). The third one are the computation

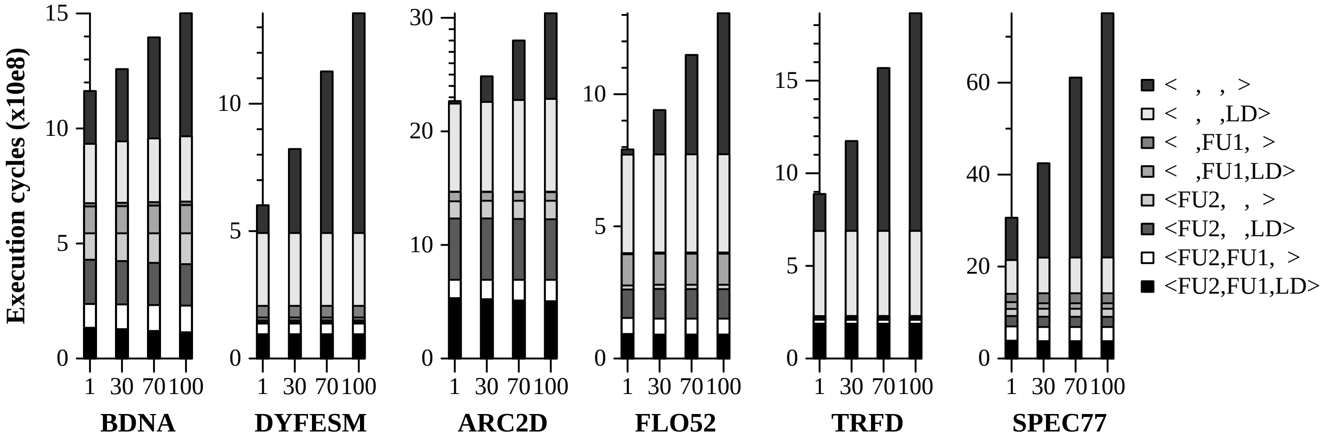

Figure 1: Functional unit usage for the reference architecture. Each bar represents the total execution time of a program for a given latency. Values on the x-axis represent memory latencies in cycles.

instructions executed in scalar mode and goes to the scalar processor (*SP*). The three processors are connected 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[9]. In order to control these three processors, the decoupled vector architecture we evaluate has a fourth processor responsible of fetching all instructions and distributing them among the *AP*, *SP* and *VP*. This processor is the fetch processor (*FP*).

All communications between processors are made through the set of queues. The naming convention used for the different queues is as follows: instruction queues are labeled using the processor name plus the suffix IQ. The other queues are data queues and use the name of their origin and destination processor to derive its name (i.e., the queue that connects the *AP* and the *VP* is called AVDQ). There are two branch queues that communicate the result of comparisons back to the fetch processor (SFBQ and AFBQ). The following sections will describe in more detail each one of the processors present in our architecture.

4.1 The Fetch Processor

The fetch processor is a very simplified version of the control unit of the reference vector architecture. It fetches instructions from a sequential, non-decoupled instruction stream, translates them into a decoupled version, and distributes these instructions to the processor responsible of executing them. The translation task is accomplished through a set of simple rules: Computation instructions are sent to their corresponding unit in a straightforward correspondence. Memory accessing instructions are sent to the *AP* and a modified version of the instruction is sent to the processor that expects to receive the data. This modified instruction (a queue mov, or QMOV for short) instructs the computation processor (either *VP* or *SP*) to move data from its input queue into a destination register. The *FP* also takes care of generating all the necessary QMOV instructions when a certain instruction requires data from another processor (typically, a vector register being operated with a scalar register). It is important to note that the QMOV's generated by the FP are not "instructions" in the real sense, i.e., they do not belong to the programmer visible instruction set. These QMOV opcodes are hidden inside the implementation.

The simulation model assumes perfect branch prediction, and, thus, the fetch processor never stalls when it encounters a jump-like instruction. This decision was made based on data presented in [6] that shows that for the six benchmark programs under study, the branch pressure is very low.

4.2 The Address Processor

The address processor performs all memory accesses, both scalar and vector, as well as all address computations. Scalar memory accesses go first through a scalar cache that holds only scalar data. Vector accesses do not go through the cache and access main memory directly. There is only one pipelined port to access memory that has to be shared by all memory accesses. The memory model assumes a common shared address bus to access memory and physically separated data paths for loads and stores. The memory model also assumes that there will be no chaining after a vector load (data can not be consumed from the AVDQ until the last element of the vector arrives from memory).

Store instructions are processed in a two-step process and are always executed in strict program order. First the effective address of the store is put in a store address queue. The *AP* has two store address queues, one for scalars and one for vectors. The scalar store address queue (`SSAQ`) holds effective addresses. The vector store address queue (`VSAQ`) needs to hold the effective address of the store plus its vector length and stride. Once a store address is entered into a queue, it will stay there until its corresponding data arrives at the store data queue. When the first slot in both an address queue and its corresponding data queue is ready, the store will be performed and the corresponding slots in the queue will be released. A vector store of length VL is considered to use the address bus for exactly VL cycles. Memory latency is not seen by the processor for stores, since this latency is paid once the address request has issued from the AP.

It is important to note that vector stores are performed "behind the back" of the AP. The AP is mainly responsible for feeding the vector store address queue. The store itself will be performed whenever there is a match between the data and the address queue. This two-step process allows the AP to proceed execution without stalling whenever it encounters a store instruction that does not have its corresponding data ready. The drawback of this scheme is that it implies the need to use dynamic memory disambiguation in order to check for possible memory hazards between loads and stores held in the queue.

Load instructions are also executed in a two-step process. First the load is disambiguated against all stores in the store queues. Disambiguation for scalar memory references is straightforward (equality test). For vector loads, disambiguation proceeds as follows. For every vector memory reference (either load or store) we have a base address BA, a vector length VL, a vector stride VS and an access granularity of S bytes. We define the *memory range* accessed by a vector reference as all memory locations comprised between BA and $BA + (VL - 1) * VS + S$ (invert this two terms for negative strides). We say there is a memory hazard between a vector load and vector store if their corresponding memory ranges overlap in at least one byte. For the special case of scatters and gathers, which can not be characterized by a memory range, the model assumes that they define all memory. Then, a vector load is checked against all memory ranges defined by stores in the `VSAQ` and the `SSAQ`. If there is a dependency with a store, the store queue contents have to be written to memory before performing the load instruction. In this case, the AP will send to memory all stores in the queue up to the youngest offending store and then resume execution and perform the stalled load. A vector load of length VL will use the address bus for exactly VL cycles and then it will release it. The first element of the vector will not arrive at the processor until L latency cycles have elapsed.

4.3 The Vector Processor

The vector processor performs all vector computations. It is almost exact to the vector part of the reference architecture described in section 2.1. The main difference between the VP and the reference architecture is that the VP has two functional units dedicated to move data in and out of the processor. This two units, the QMOV units, are able to move data from the AVDQ data queue (filled by AP) into the vector registers and move data from the registers into the VADQ (which will be drained by AP sending its contents to memory). We have included *two* QMOV units instead of one because otherwise the VP would be paying a high overhead in some very common sequences of code, when compared to the reference architecture.

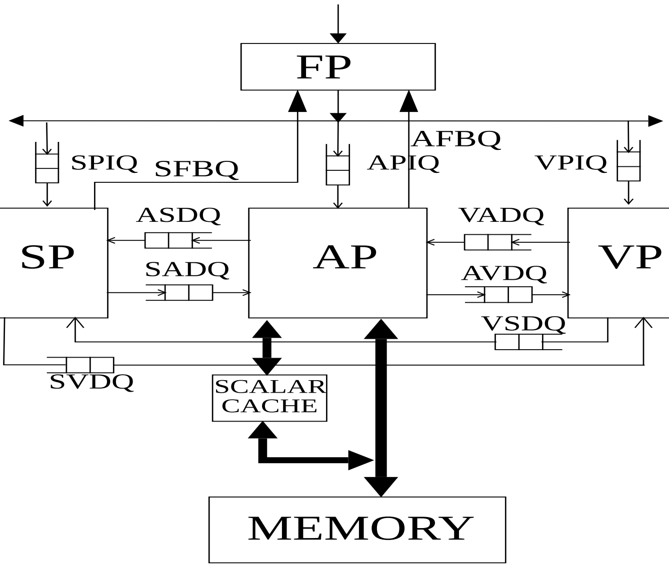

Figure 2: The decoupled vector architecture studied in this paper.

4.4 The Scalar Processor

In order to better understand the behavior of the vector component of our vector architecture and given the high variation in designs in the scalar cpus of current vector machines, we have decided to use a very simplistic model for our scalar processor. This model states that the scalar processor issues only one instruction per cycle and that all scalar instructions complete in exactly on cycle. The only exceptions to this rule are those instructions that keep the synchronization with the other processors. That is, `QMOV` instructions executed in the *SP are* blocking instructions that may result in a stall if the queue used is either empty or full.

5 Performance of the Decoupled Vector Architecture

In this section we present the performance of the decoupled vector architecture versus the reference architecture. In order to compare the effectiveness of both architectures in executing vector programs, we have run simulations for each of the benchmark programs both on the reference architecture simulator and the decoupled vector architecture simulator. At each run, the only changing parameter is the memory latency. We have also included in the results lower bounds for the execution time of each program. Figure 3 presents the results of these simulations for the selected programs.

To compute the lower bound for one of the programs we consider what would be the execution time if there were no dependencies at all. We consider only

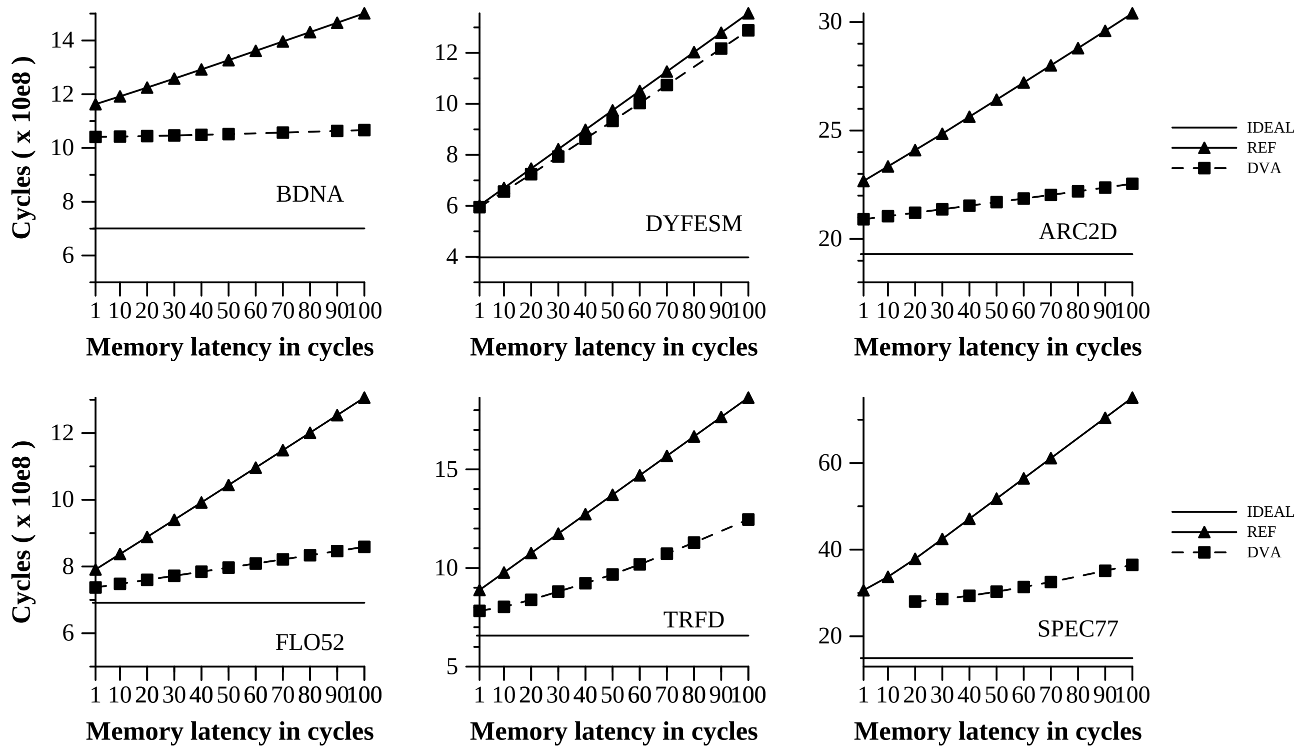

Figure 3: DVA versus Reference architecture for the benchmark programs

resource constraints in order to determine the minimum possible execution time. Given that our two architectures both have essentially five resources (unit `FU1`, unit `FU2`, the memory port, the scalar processor and the scalar cache), we partition all operations executed by a program into these five categories. Then, the category that has the maximum number of operation determines the minimum theoretical execution time for the program.

For the decoupled simulations, we have used the following parameters: instruction queues were all 16 instructions long. All scalar queues were of length 256. The vector load queue (AVDQ) also had 256 slots, where each slot is a vector register. The vector store queue (VADQ) was set to 16 slots. The rationale for these values is as follows: all queues were first set at "infinite values", that is, 512 slots for the instruction queues and 256 slots for all other queues, to find a bound to the speedup achievable by decoupling. Then, simulations were conducted to evaluate the effect of reducing the sizes of these queues. For the instruction queues, simulations showed that reducing their length to 16 slots did not affect noticeably final performance (less than 2% difference). For the vector store queue, the simulations showed that there is almost no difference between 16, 32 and 256 slots. In order to bound the search space of possible configurations, in this paper we have set the store queue to 16 elements for all experiments. See [5] for results on the other sizes of the store queue. For the vector load queue we will present results using a queue length of 256 and next section will present data on the actual usage of the queue.

The overall results suggest two important points. First, the DVA architecture shows a clear speedup over the REF architecture even when memory latency is just 1 cycle. Even if there is no latency in the memory system, the decoupling produces a similar effect as a prefetching technique, with the advantage that the AP knows which data has to be loaded (no incorrect prefetches). The second important point is that the slopes of the execution time curves for the reference and the decoupled architectures are substantially different. This implies that decoupling tolerates long memory delay much better than current vector architectures.

Overall, decoupling is helping to minimize the number of cycles where the machine is halted waiting for memory. Recall from section 3 that the execution time of the program could be partitioned into eight different states. Decoupling greatly reduces the cycles spent in state $\langle\ ,\ ,\ \rangle$. Figure 4 illustrates this point. This figure presents the ratio between the number of cycles spent in state $\langle\ ,\ ,\ \rangle$ in the REF architecture and the number of cycles spent in state $\langle\ ,\ ,\ \rangle$ in the DVA architecture. As it can be seen, the reduction in stall cycles can be as high as a factor of 5 to 1 (ARC2D). Programs FLO52 and SPEC77 show also large reductions, over 4 and 3 to 1 respectively.

To summarize the speedups obtained, figure 5 presents the speedup of the DVA over the REF architecture for each particular value of memory latency. Speedups (at latency 100) range from a 1.35 for ARC2D to a 2.05 for SPEC77. The only program that does not show significant speedups, DYFESM, is not affected negatively by the decoupling principle.

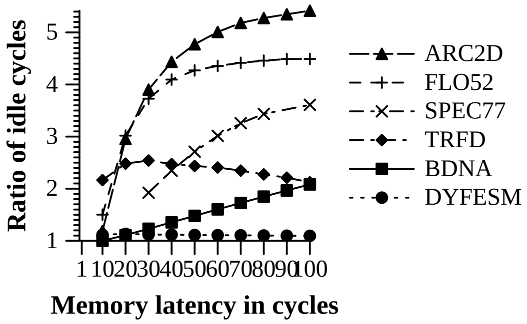

Figure 4: Ratio of cycles spent in state ⟨ , , ⟩ between the REF and the DVA architectures.

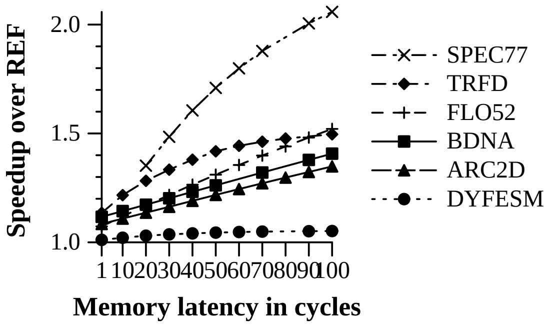

Figure 5: Speedup of the DVA over the Reference architecture for the benchmark programs

We have investigated DYFESM and we found out that its three most important loops do not benefit from decoupling. The first loop, which is responsible for 68% of all vector operations, can not be executed in less than 3 chimes, and both the reference and the decoupled architectures achieve this minimum. Thus, it does not show any speedup. The next two loops, each one responsible for 7.1% of all vector operations, have a reduction vector operation that has a dependency with itself of distance 1. This dependency makes the *SP* stall and prevents the *AP* (which is waiting for a register coming from *SP*) to get ahead of the *VP*. The three processors have to work in a lockstep fashion and can not improve upon the reference architecture.

6 Length of the Vector Queues

The previous section has used a load queue (AVDQ) length of 256 elements. In this section we will study the actual usage of this queue. As already mentioned in the previous section, the store queue was set in all experiments at 16 slots.

Figure 6 presents the distribution of busy slots in the AVDQ for the benchmark programs. For each program we plot three distributions corresponding to three different memory latency values. Each bar in the graphs represents the total number of cycles that the AVDQ had a certain number of busy slots (We plot absolute number of cycles instead of percentages to be able to compare the three different latencies). For example, for BDNA, the AVDQ was completely empty (zero busy slots) for more than 400 millions of cycles.

From figure 6 we can see that for the six benchmark programs it is uncommon to use more than four slots in the queue. At latency 1, most programs have typically 0 or 1 busy slots. When moving to latency 30, the graphs show a clear increase in the number of cycles with 2 busy slots. At latency 100, the highest percentages are with 2 busy slots and the number of cycles with 3, 4 or even 5 busy slots become important. As expected, the longer the memory latency, the higher the number of busy slots, since the memory system has more outstanding requests and, therefore, needs more slots in the queue. The sharp increase in the number of cycles having 2 busy slots can be explained by analyzing the characteristics of the programs. The six benchmarks are, as a whole, memory bound. Therefore, it is fair to assume that the majority of their loops will also be memory bound. When the DVA is executing a memory bound loop, what happens is that the *VP* executes faster than the *AP*. In the steady state, the *VP* will most of the time be waiting for data to arrive to its input queue (the AVDQ). As soon as one vector is ready in the AVDQ, the *VP* will execute a `QMOV` instruction and start moving it to a vector register. At the same time the *AP* will most probably start another vector load. Thus, we will have two busy slots in the AVDQ.

Another important point is that the queue length seems to be bounded by 9 slots, with none of the programs having at any point in time more than 8 full slots. This is a counterintuitive result, since one would expect that, for compute bound loops, the AVDQ would be completely filled. When a loop is compute bound, the resource that becomes the bottleneck is the VPIQ. The VPIQ is limited to 16 slots, and for each memory operation the *FP* will insert a QMOV pseudo-instruction into the VPIQ. Thus, a compute bound loop will be able to hold a maximum of 9 computation instructions and 7 QMOV's in the VPIQ. Once the VPIQ is full, the *FP* will block and the *AP* will not be able to get more vector loads. As a result, the *AP* will not be able to insert more than eight elements in the AVDQ. Note that this behavior will not affect final performance since, if the loop is compute bound, the maximum speed is determined by the *VP* and not by the *AP*.

7 Bypass

When a vector load instruction is about to be issued, the DVA disambiguates it against all stores held in the vector store queue. If there is a dependency, the store queue contents must be sent to main memory before proceeding with the load. An interesting possibility is that the load might be *identical* to some store in the store queue. In such a case, we might do a bypass between the VADQ and the AVDQ and discard the load. This bypass, which would take `VL` cycles since it would be performed in the processor itself, would be much cheaper than the corresponding memory access. On top of that, it would reduce

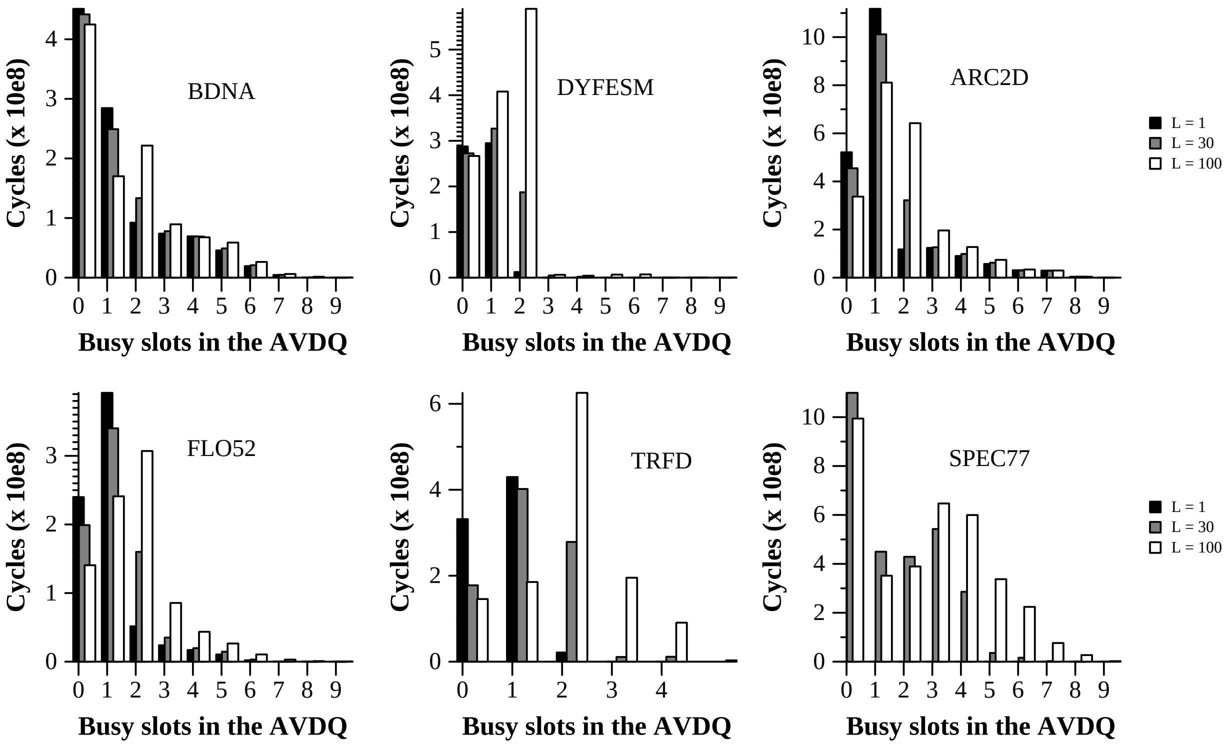

Figure 6: Busy slots in the AVDQ for the benchmark programs for three different memory latency values.

the total memory traffic, since the load is performed without accessing main memory.

This bypassing is a limited form of data caching and has several advantages. First, the data being bypassed does not suffer any memory latency penalties. Second, during the bypass operation the memory port is idle and can be used by subsequent independent vector memory operations. In this second case, bypassing gives the illusion of having two memory ports, since two different vectors are being moved into the AVDQ simultaneously.

When will this type of bypassing occur ? We have two possibilities. The first possibility is that this bypassing can occur between data belonging to the same iteration of a vector loop. Most probably, this data being stored and reloaded in the same iteration will be spill code inserted by the compiler. The same kind of storing and reloading vector data also happens at procedure call/return boundaries. The second possibility is that we have a bypass between data belonging to different iterations of the same loop. We believe that bypassing will be specially useful in the first case described. In [6] it is shown that the six programs have a considerable amount of spill code (in BDNA 69.5% of all memory operations are spill loads and stores; in ARC2D 12.2%, in FLO52 11.9%, and in SPEC77 3%) and thus we expect these programs to benefit from the bypassing mechanism.

A question raised by the bypassing technique is the appropriate length for the vector store queue. The goal is to be able to determine the store queue length so that the vast majority of the vector spill code can be captured by the bypass mechanism. To study the effects of bypassing we will fix the load queue length to four slots (the value suggested in section 6) and we will only vary the store queue length.

Figure 7 presents the comparison, for the six benchmark programs, between the DVA and four Bypass configurations: BYP 4/4, BYP 4/8, BYP 4/16 and, as a lower bound, BYP 256/16 (the number on the left is the load queue length and the number on the right is the store queue length). The graphs show how all bypass configurations are always better than their DVA counterpart (except in the SPEC77 case). A positive result is that even at memory latency of 1 cycle the speedups are significant: if we compare the BYP 256/16 with the DVA, DYFESM and TRFD lead with a 22.0% and a 17.36% speedup, followed by BDNA (10.94%) and FLO52(9.31%). For ARC2D the gains are lower (2.68%) and for SPEC77 are close to zero (0.7%). Since the cost of retrieving something from memory at latency 1 is the same as the cost of performing a bypass, this speedups at latency 1 show that the bypass unit is in fact acting as a second "memory port". The ability to have an outstanding bypass and a memory access allows that for long periods of time the *AP* appears to have to different paths to memory. This is confirmed by the striking result for FLO52, where the bypass version is able to outperform its theoretical lower bound! This is due to the fact that the lower bound is computed assuming a single memory port, and does not account for the possibility of bypassing.

The effect of varying the store queue length can also be observed in figure 7. In three cases (DYFESM, ARC2D and TRFD), the difference between a store

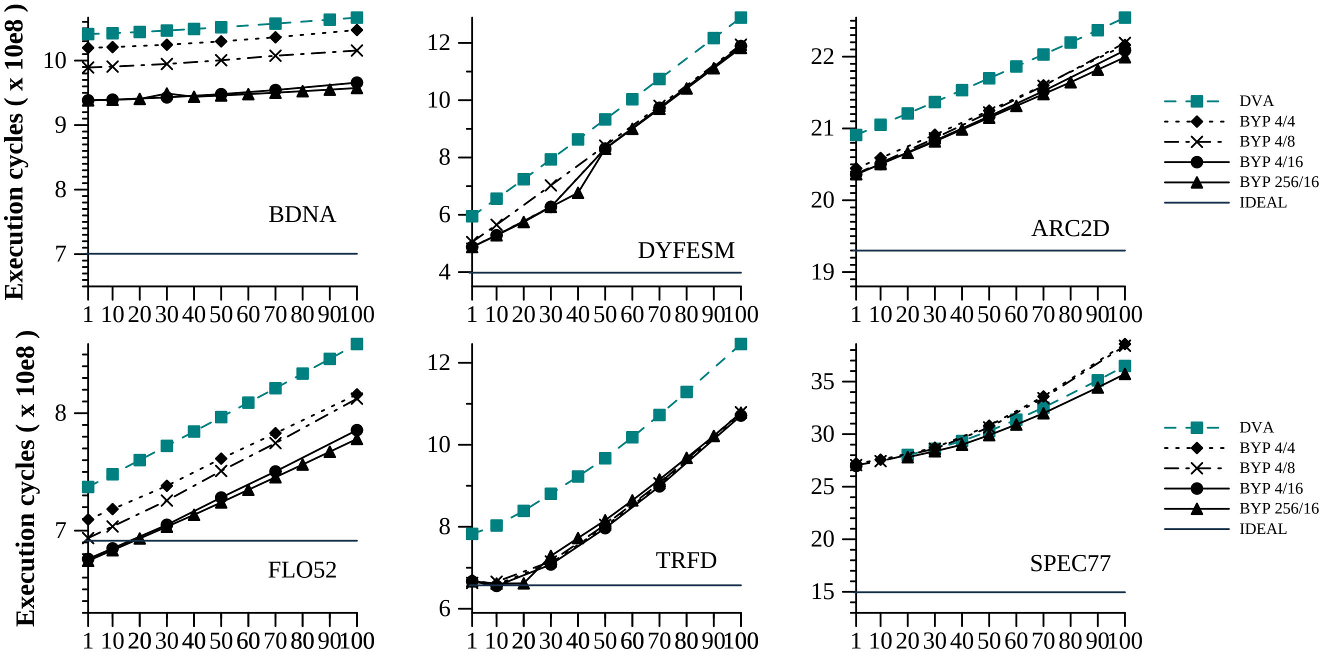

Figure 7: Performance of the Bypassing scheme. X-axis is memory latency in cycles.

queue length of 4, 8 or 16 slots is under 2%. For BDNA and FLO52, the BYP 4/8 is only a 5.4% and a 2.6% away from the lower bound represented by the BYP 256/16. Thus, a store queue length of 8 elements seems enough to capture more than 95% of the performance of a longer queue of sixteen elements. Whether increasing the queue length further is beneficial or not is a subject that deserves further study and that we are pursuing right now.

The SPEC77 programs is a special case. The three bypass configurations that have a load queue length of four are worse than the DVA configuration. This has nothing to do with bypassing. Recall that the DVA configuration has 256 load slots. The SPEC77 program makes a heavy use of these slots (see figure 6 and thus reducing the load queue to only four slots slows down the program. See how, when the two load queues are set at the same size (BYP 256/16 configuration) bypass does show some improvement over the DVA.

The key point is that bypassing is effective with moderately sized queues. For the store queue, in almost all programs eight slots achieve the same performance as the 16 slots queue, and seem to capture the majority of the spill code present in the programs. For the load queue, figure 7 shows how the difference between a four slots queue and an "infinite" (256) queue is very small.

Apart from the reduction in execution time, the other interesting effect of bypassing is the reduction in memory traffic. Since when a vector is bypassed from the store queue to the load queue, the *AP* does not access main memory at all, each bypass translates into a net reduction of memory traffic. Figure 8 compares the total memory traffic in the BYP 256/16 architecture to the total memory traffic in the DVA architecture. As it can be seen, the reduction is quite high. For DYFESM and TRFD it is over 30% and for BDNA and FLO52 it is around 10%.

8 Conclusions and Future Work

In this paper we have presented decoupled vector architectures. We have described a basic decoupled architecture (DVA) that uses the principles of decoupling to hide most of the memory latency seen by vector processors.

The DVA architecture shows a clear speedup over the REF architecture even when memory latency is just 1 cycle. This speedup is due to the fact that the *AP* slips ahead of the *VP* and loads data in advance, so that when the *VP* needs its input operand they are (almost) always ready in the queues. Even if there is no latency in the memory system, this "slipping" produces a similar effect as a prefetching technique, with the advantage that the *AP* knows which data has to be loaded (no incorrect prefetches). Thus, the partitioning of the program into separate tasks helps in exploiting more parallelism between the *AP* and *VP* and translates into an increase in performance, even in the absence of memory latency. Moreover, as we increase latency, we see how the slopes of the curves of the execution time of the benchmarks remain fairly stable, whereas the REF architecture is much more sensitive to the increase in memory latency.

We have introduced a bypassing technique that allows to copy data from the vector store queue back

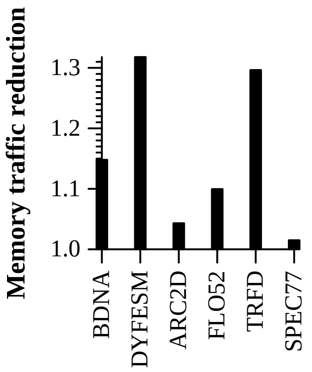

Figure 8: Ratio of total memory traffic between the DVA 256/16 and the BYP 256/16 architectures.

into the vector load queue. This bypassing is able to service some memory requests without paying the main memory latency and to reduce the total amount of memory traffic. We have shown how when this bypassing is used in the decoupled architecture, there is an average speedup of 10% over the DVA architecture.

Finally, we have seen that this speed improvements can be implemented with a reasonable cost/performance tradeoff. Section 7 has shown how the length of the queues does not need to be very large to allow for the decoupling to take place. A vector load queue of four slots is enough to achieve a high fraction of the maximum performance obtainable by an infinite queue. On the other side, the vector store queue does not need to be very large. Our experiments varying the store queue length indicate that a store queue of eight elements achieves almost the same performance as one with sixteen slots.

We are now currently working in the comparison of decoupling with techniques such as out-of-order execution and register renaming. We are also extending the studies on queue length.

Acknowledgments

We would like to thank specially Prof. James E. Smith for his comments on an earlier draft of this paper and the anonymous referees for extensive reviews that helped improving the presentation of this work.

References

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

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

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

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

[5] R. Espasa and M. Valero. Decoupled vector architectures. Technical report, UPC, 1994.

[6] R. Espasa and M. Valero. Instruction level characterization of the Perfect Club programs on a vector computer. Technical report, UPC-CEPBA-1995-12, 1995.

[7] 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.

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

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

[10] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *ICS*, pages 323–334, July 1992.

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

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

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

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

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

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

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

[18] J. E. Smith, G. Dermer, B. Vanderwarn, S. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon. The ZS-1 central processor. In *ASPLOS-2*, pages 199–204. CS press, 1987.

[19] J. E. Smith, A. R. Pleszkun, R. H. Katz, and J. R. Goodman. PIPE: A high-performance VLSI architecture. In *IEEE International Workshop on Computer System Organiztion*, March 1983.

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

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

[22] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, 1995.