Journal of Supercomputing, , 1–25 ()

A Simulation Study of Decoupled Vector Architectures

ROGER ESPASA AND MATEO VALERO {roger,mateo}@ac.upc.es
Dept. Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona

Editor:

Abstract. The purpose of this paper is to show that decoupling techniques can be applied to a vector processor, resulting in a large increase in performance of vectorizable programs. Using a trace driven approach, we simulate a selection of the Perfect Club and Specfp92 programs and compare their execution time on a conventional single port vector architecture and on a decoupled vector architecture. Decoupling provides a performance advantage of more than a factor of 1.4 for realistic memory latencies, and even with an ideal memory system with zero latency, there is still a speedup of as much as 1.31. An important part of this paper is devoted to study the tradeoffs involved in choosing an suitable size for the different queues of the architecture, so that the hardware cost of the queues is reduced while still retaining most of the performance advantages of decoupling.

Keywords: Vector architectures, decoupling, instruction-level parallelism, memory latency

1. Introduction

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

Current superscalar processors have been attacking the memory latency problem through basically three main types of techniques: caching, 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 [5, 23] 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 [16] have proven very useful to fit the working set of a program into the cache. Recently, address and data prediction receive much attention as a potential solution for indirectly masking memory latency [21].

Multithreaded processors [1, 30] 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 [27, 25, 18] 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. 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 [24, 8] show how the performance achieved by vector architectures on real programs is far from the theoretical peak performance of the machine. In [8] it 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 [17, 19, 28, 22] 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. Many techniques can be borrowed from the superscalar microprocessor world. In this paper we focus on decoupling, but we have also explored other alternatives such as multithreading [12] and out-of-order execution [13].

The purpose of this paper is to show that using decoupling techniques in a vector processor [11], the performance of vector programs can be greatly improved. We will show how, even for an ideal memory system with zero 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. Another benefit of decoupling is that it also allows to tolerate latencies inside the processor, such as functional unit and register crossbar latencies.

This paper is organized as follows. Section 2 describes both the baseline and decoupled architectures studied throughout this paper. In section 3 we discuss our simulation environment and the benchmark programs used in the experiments presented. Section 4 provides a background analysis of the performance of a traditional vector machine. In section 5 we detail the performance of our decoupled vector proposal. Finally, section 6 presents our conclusions and future lines of work.

2. Vector Architectures and Implementations

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

The main implication of the election of a C3400 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.

2.1. The Reference Architecture

The Convex C3400 [7] consists of a scalar unit and an independent vector unit (see fig. 1). 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.

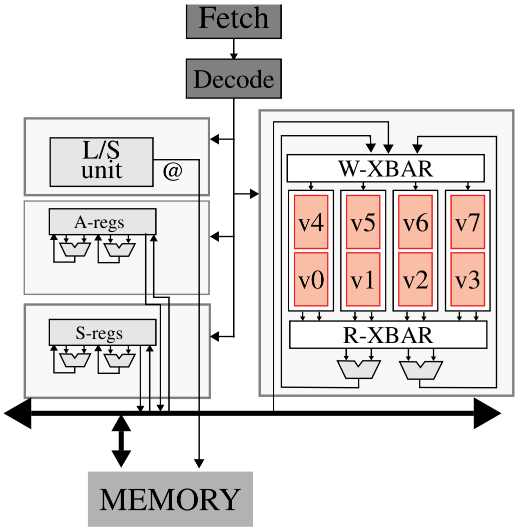

Figure 1. The reference vector architecture modeled after a Convex C3400.

2.2. The Decoupled Vector Architecture

The decoupled vector architecture we propose uses a fetch processor to split the incoming, non-decoupled, instruction stream into three different decoupled streams (see fig. 2). 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[15]. The main difference of this decoupled architecture with previous scalar decoupled architectures such as the ZS-1 [26], the MAP-200 [6], PIPE [14] or FOM [4], 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 fetch processor fetches instructions from a sequential, non-decoupled instruction stream and translates them into a decoupled version. The translation is such that each processor can proceed independently and, yet, synchronizes through the communication queues when needed. For example, when a memory instruction that loads register v5 is fetched by the *FP*, it is translated into two pseudo-instructions: a load instruction, which is sent to the *AP*, that will load data into the vector load data queue (VLDQ, queue no. 1 in fig. 2), and a **qmov** instruction, sent to the *VP*, that dictates a move operation between the VLDQ and the final destination register v5. It is important to note that the **qmov**'s generated by the FP are not

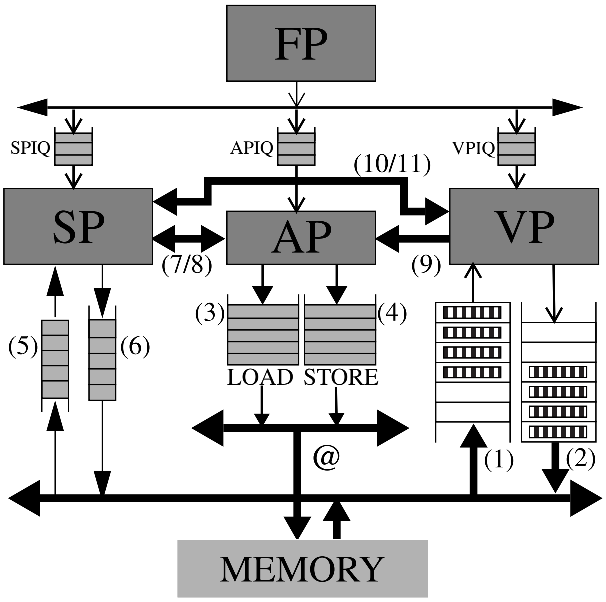

Figure 2. The decoupled vector architecture studied in this paper. Queue names: (1) vector load data queue –VLDQ, (2) vector store data queue –VSDQ, (3) address load queue –ALQ, (4) address store queue –ASQ, (5) scalar load data queue –SLDQ, (6) scalar store data queue –SSDQ, (7/8) Scalar-Address Control Queues, (9) Vector-Address Control Queue, (10/11) Scalar-Vector Control Queues

"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.

Note that the total hardware added to the original reference architecture shown in figure 1 consists only of the communication queues and a private decode unit for each one of the three processors. The resources inside each processor are the same in the decoupled vector architecture and in the reference architecture. It is worth noting, though, that while most queues added are scalar queues and, therefore, require a small amount of extra area, the VLDQ and VSDQ hold full vector registers (queues 2 and 3 in Fig. 2). That is, each slot in these queues is equivalent to a normal vector register of 128 elements, thus requiring 1Kb of storage space. One of the key points in this architecture will be to achieve good performance with relatively few slots in these two queues.

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 address processor inserts load instructions into the Address Load Queue (ALQ) and store instructions into the Address Store Queue (ASQ). Stores stay in the queue until their associated data shows up either in the output queue of the *VP* (the vector store data queue – VSDQ), or in the output queue of the *SP* (the scalar store data queue –SSDQ). When either a load or a store becomes ready, i.e, it has no dependencies and its associated data, if necessary, is present, it is sent over the address bus as soon as it becomes available. In the case of having both a load and a store ready, the *AP* always gives priority to loads.

To preserve the sequential semantics of a program, the address processor needs to ensure a safe ordering between the memory instructions held in the ALQ and ASQ. All memory accesses are processed in two steps: first, their associated "memory region" is computed. Second, this region is used to disambiguate the memory instruction against all previous memory instructions still being held in the address queues of the *AP*. Using this disambiguation information, a dependency scoreboard is maintained. This scoreboard ensures that (1) all loads are executed in-order, (2) all stores are executed in-order and (3) loads can execute before older stores if their associated memory regions do not overlap at all. When dependences are found, the scoreboard guarantees that loads and stores will be performed in the original program order so that correctness is guaranteed.

A "memory region" is defined by a 5-tuple: $\langle @_1, @_2, \mathtt{vl}, \mathtt{vs}, \mathtt{sz}\rangle$. Virtual addresses $@_1$ and $@_2$ are the start and end addresses, resp., of a consecutive region of bytes in memory, and vl, vs, and sz are the vector length, vector stride and access granularity needed by vector memory operations. The end address, $@_2$, is computed as $@_1 + (vl-1) * vs + sz$. For scalar memory accesses, vl is set to 1 and vs to 0. For the special case of gathers and scatters, which can not be properly characterized by a memory region, $@_1$ is set to 0 and $@_2$ is set to $2^{32}-1$, so that the scoreboard will find a dependence between a gather/scatter and all previous and future memory instructions.

The vector processor performs all vector computations. 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 VLDQ 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.

The set of control queues connecting the three processors, queues 7–11 in Fig. 2, are needed for those instructions that have mixed operands. The most common case is in vector instructions, that can have a scalar register as a source operand (i.e., `mul v0,s3 -> v5`). Other cases include mixed A- and S- register instructions, vector gathers that require an address vector to be sent to the *AP*, and vector reductions that produce a scalar register as a result.

3. Methodology

3.1. Simulation Environment

To asses the performance benefits of decoupled vector architectures we have taken a trace driven approach. The Perfect Club and Specfp92 programs have been chosen as our benchmarks [3]. The tracing procedure is as follows: the Perfect Club programs are compiled on a Convex C3480 [7] 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 [9], 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.

3.2. The benchmark programs

Because we are interested in the benefits of decoupling for vector architectures, we selected benchmark programs that are highly vectorizable ($\geq$ 70%). From all programs in the Perfect and Specfp92 benchmarks we chose the 10 programs that achieve at least 70% vectorization. Table 1 presents some statistics for the selected Perfect Club and Specfp92 programs. Column number 2 indicates to what suite each program belongs. Column 3 presents the total number of memory accesses, including vector and scalar and load and store accesses. Next column is the total number of operations performed in vector mode. Column 5 is the number of scalar instructions executed. 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. Finally column seven presents the average vector length used by vector instructions, and is the ratio of vector operations over vector instructions.

Table 1. Basic operation counts for the Perfect Club and Specfp92 programs (Columns 3–5 are in millions).

Program	Suite	Mem Ops	Vect Ops	Scal Ins	% Vect	avg. VL
`swm256`	Spec	4499	5036	4	99.9	127
`hydro2d`	Spec	1785	2203	23	99.0	101
`arc2d`	Perf.	1959	2157	30	98.5	95
`flo52`	Perf.	706	551	18	97.1	54
`nasa7`	Spec	1795	2190	64	96.2	58
`su2cor`	Spec	1561	1862	66	95.7	125
`tomcatv`	Spec	527	403	92	87.9	127
`bdna`	Perf.	795	889	128	86.9	81
`trfd`	Perf.	826	438	156	75.7	22
`dyfesm`	Perf.	502	298	108	74.7	21

The most important thing to remark from table 1 is that all our programs are memory bound when run on the reference machine. If we take column labeled "Vect Ops" and divide it by 2 we get the minimum number of cycles required to execute all vector computations on the two vector functional units available. Comparing now column "Mem Ops" against the result of our division we will see that the bottleneck for all these programs is always the memory port. That is, the absolute minimum execution time for each of these programs is determined by the total amount of memory accesses it performs.

This remark is worth keeping in mind, since, as following sections will show, even if the memory port is the bottleneck for all programs, its usage is not always as good as one would intuitively expect.

4. Bottlenecks in the Reference Architecture

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

Consider only the three vector functional units of our reference architecture (**FU2**, **FU1** and **MEM**). 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 $\langle FU2, FU1, MEM\rangle$ represents a state where all units are working, while $\langle\ ,\ ,\ \rangle$ represents a state where all vector units are idle. The execution time of a program can thus be split into eight possible states.

Figure 3 presents the splitting of the execution time into states for the ten benchmark programs. We have plotted the time spent in each state for memory latencies of 1, 20, 70, and 100 cycles. From this figure we can see that the number of cycles where the programs proceed at peak floating point speed (states $\langle FU2, FU1, MEM\rangle$ and $\langle FU2, FU1,\ \rangle$) is quite low. The number of cycles in these states changes relatively little as the memory latency increases, so the fraction of

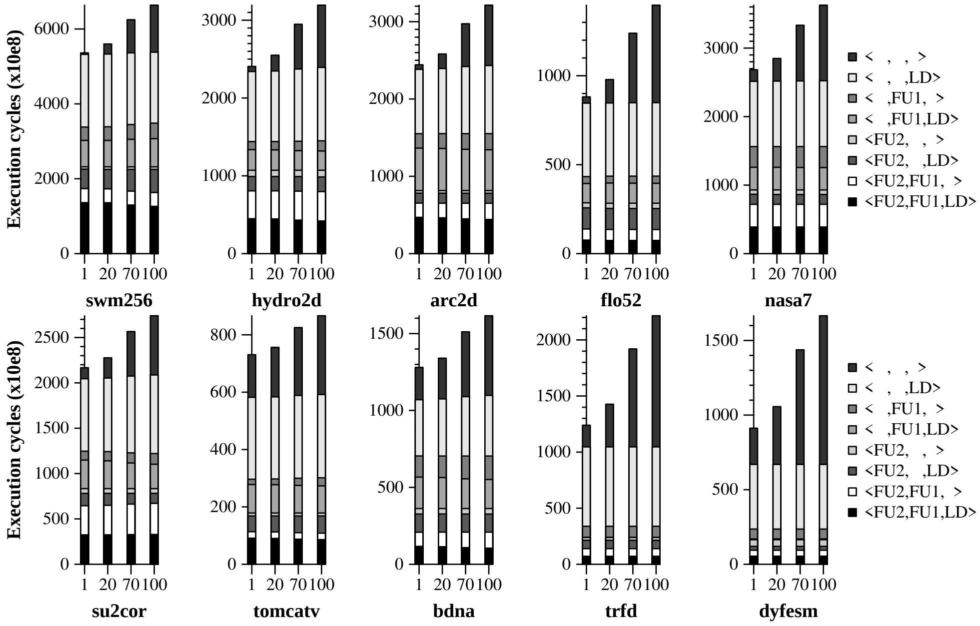

Figure 3. Functional unit usage for the reference architecture. Each bar represents the total execution time of a program for a given latency. Values on the x-axis represent memory latencies in cycles.

fully used cycles decreases. Memory latency has a high impact on total execution time for programs `dyfesm`, and `trfd` and `flo52`, which have relatively small vector lengths. The effect of memory latency can be seen by noting the increase in cycles spent in state $\langle\ ,\ ,\ \rangle$.

The sum of cycles corresponding to states where the MEM unit is idle is quite high in all programs. These four states ($\langle\ ,\ ,\ \rangle$, $\langle\ , FU1,\ \rangle$, $\langle FU2,\ ,\ \rangle$and $\langle FU2, FU1,\ \rangle$) correspond to cycles where the memory port could potentially be used to fetch data from memory for future computations. Figure 4 presents the percentage of these cycles over total execution time. At latency 70, the port idle time ranges between 30% and 65% of total execution time. All 10 benchmark programs are memory bound when run on a single port vector machine with two functional units. Therefore, these unused memory cycles are not the result of a lack of load/store work to be done.

5. Performance of the DVA

In this section we present the performance of the decoupled vector architecture versus the reference architecture (REF). We first start by ignoring all latencies of the functional units inside the processor and concentrate on the study of the effects of main memory latency (sections 5.1–5.5). This study will determine the most

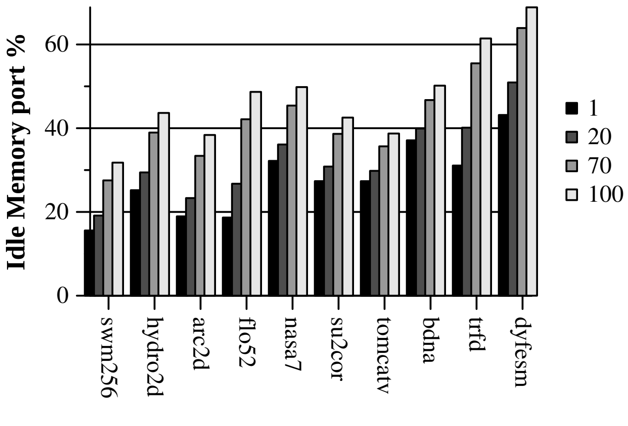

Figure 4. Percentage of cycles where the memory port was idle, for 4 different memory latencies.

cost-effective parameters that achieve the highest performance. Then we proceed to consider the effect of arithmetic functional units and register crossbar latencies in execution time (section 5.6). We will first show that decoupling tolerates very well memory latencies and is also useful for tolerating the smaller latencies inside the processor.

We will start by defining a DVA architecture with infinite queues and no latency delays –the Unbounded DVA, or UDVA for short– that we will compare to the reference architecture. Then we will introduce limitations into the UDVA, such as branch misprediction penalties, limited queue sizes and real functional unit latencies, step by step to see the individual effect of each restriction. After all these steps we will reach a realistic version of the DVA – the RDVA– that will be compared against the REF and UDVA machines.

5.1. UDVA versus REF

The Unbounded DVA architecture (UDVA) is a version of the decoupled architecture that has all of its queues set to a very large value (128 slots) and no latency delays. Moreover, a perfect branch prediction model is assumed. The I-cache is not modeled in any of the following experiments, since our previous data indicates a very low pressure on the I-cache [10]. All arithmetic functional units, both scalar an vector, have a 1 cycle latency. The vector register file read and write crossbars have no latency and there is no startup penalty for vector instructions.

The benefits of decoupling can be seen in fig. 5. For each program we plot the total execution time of the UDVA and the REF architectures when memory latency is varied between 1 and 100 cycles.

In each graph we also show the minimum absolute execution time that can theoretically be achieved (curve “IDEAL”, along the bottom of each graph). To compute the IDEAL execution time for a program we use the total number of cycles con-

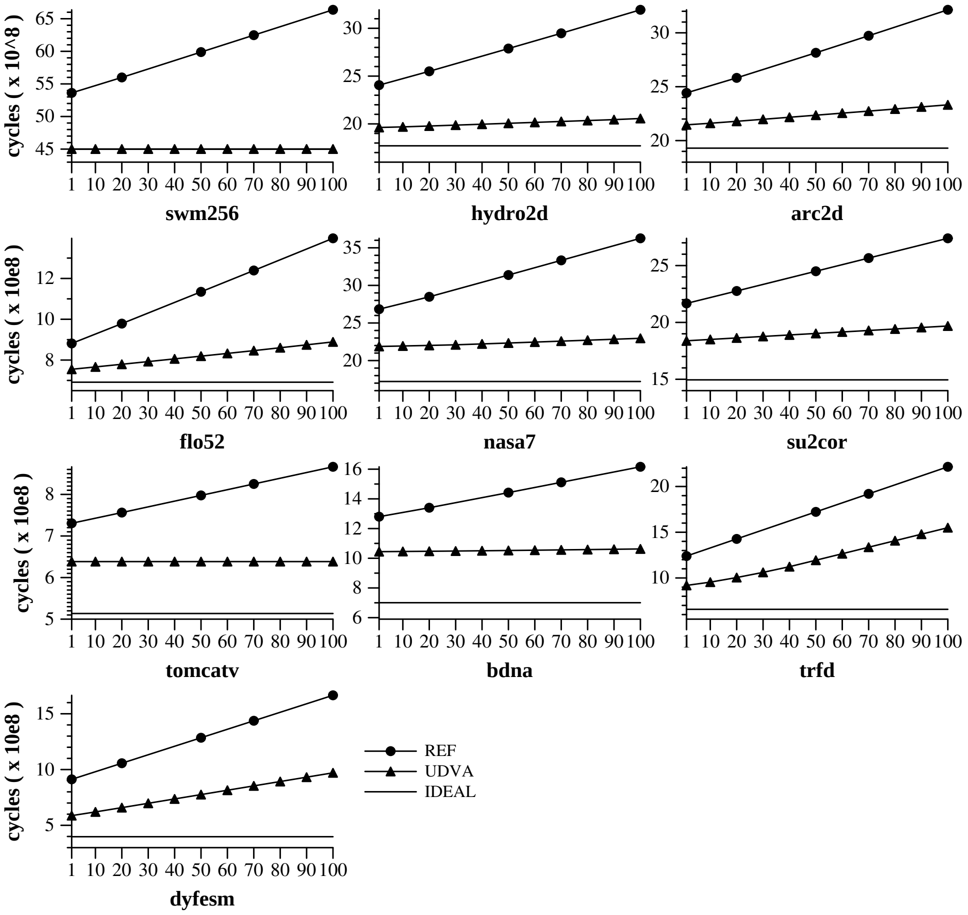

Figure 5. UDVA versus Reference architecture for the benchmark programs.

sumed by the most heavily used vector unit (FU1, FU2, or MEM). Thus, in IDEAL we essentially eliminate all data and memory dependences from the program, and consider performance limited only by the most saturated resource across the entire execution.

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.

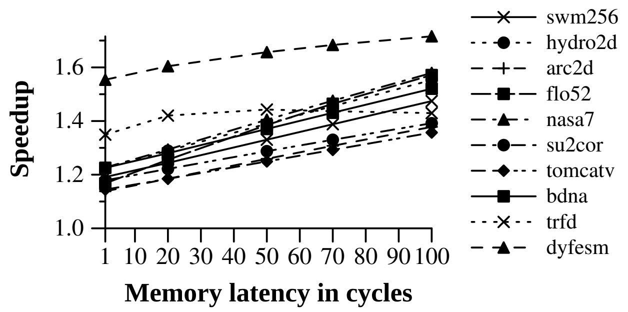

Figure 6. Speedup of the DVA over the Reference architecture for the benchmark programs

Overall, decoupling is helping to minimize the number of cycles where the machine is halted waiting for memory. Recall from section 4 that the execution time of the program could be partitioned into eight different states. Decoupling greatly reduces the cycles spent in state $\langle\ ,\ ,\ \rangle$.

To summarize the speedups obtained, fig. 6 presents the speedup of the DVA over the REF architecture for each particular value of memory latency. Speedups (at latency 100) range from a 1.32 for TOMCATV to a 1.70 for DYFESM.

5.2. *Reducing IQ length*

The first limitation we introduce into the UDVA is the reduction of the instruction queues that feed the three computational processors (*AP*, *SP*, *VP*). In this section we look at the slowdown experienced by the UDVA when the size of the APIQ, SPIQ and VPIQ queues is reduced from 128 instructions to 32, 16, 8 and 4 instructions only. In order to reduce the amount of simulation required, we have chosen to fix the value of the memory latency parameter at 50 cycles. As we have seen in the previous section, the UDVA tolerates very well a wide range of memory latencies. Thus we expect this value of 50 cycles to be quite representative of the full 1–100 latency range.

The size of the instruction queues is very important since it gives an upper bound on the occupation of all the queues in the system. For example, it determines the maximum number of entries that we can have waiting in the load address queues.

Figure 7 presents the slowdown with respect to the UDVA for our ten benchmarks when the three instruction queues are reduced to 32, 16, 8 and 4 slots. From fig. 7 we can see that the performance for 128-, 32- and 16-entries instruction queues is virtually the same for all benchmarks. From these numbers, we decided to set the IQ length to 16 entries for the rest of experiments presented in this paper. This size is in line with the typical instruction queues found in current microprocessors [31].

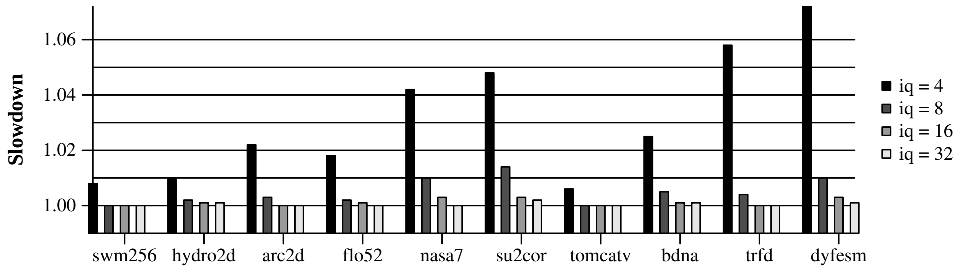

Figure 7. Slowdown experienced by UDVA when reducing the IQ size.

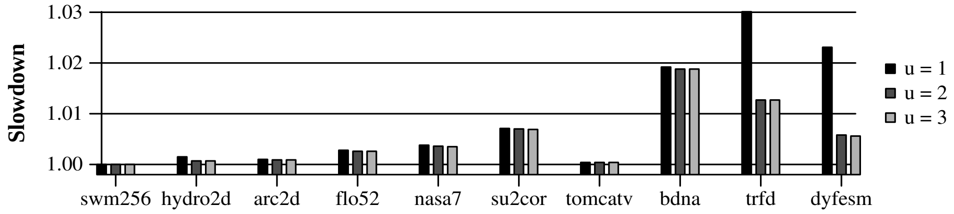

Figure 8. Slowdown due to branch mispredictions for three models of speculation.

5.3. Effects of branch prediction

In this section we look at the negative effects introduced by branch mispredictions. The branch prediction mechanism evaluated is a direct-mapped BTB holding for each entry the branch target address and a 2-bit predictor (the predictor found in [20]). We augmented the basic BTB mechanism with an 8-deep return stack (akin to the one found in [2]).

We evaluated the accuracy of the branch predictor for a 64 entries BTB. The accuracy varies a lot across the set of benchmarks. Programs FLO52 and NASA7 come out with the worst misprediction rate (around 30%) while TOMCATV has less than 0.4% of mispredicted branches. Nonetheless, the misprediction rate is rather high for a set of programs that are considered to have an "easy" jumping pattern (numerical codes tend to be dominated by DO-loops). This is due to the combination of two facts: first, vectorization has reduced the absolute total number of branches performed by the programs in an unbalanced way. The number of easily-predictable loop branches has been diminished by a factor that is proportional to the vector length (could be as high as 128) while the difficult branches found in the remaining scalar portion of the code are essentially the same. The second factor is that we are using a very small BTB compared to what can be found in current superscalar microprocessors, where a typical BTB could have up to 4096 entries [29].

Although the prediction accuracy is not very good, the impact of mispredicted branches on total execution time is very small. Figure 8 presents the slowdown due to mispredicted branches relative to the performance of the architecture from section 5.2. Since the prediction accuracy was not very high, we tested the benefit that could be obtained by being able to speculate across several branches. In fig. 8 the bars labeled "u=1" correspond to an architecture that only allows one unresolved branch. Bars labeled "u=2" and "u=3" correspond to being able to speculate across 2 and 3 branches respectively.

A first observation is that the impact of mispredicted branches is rather low. See how while FLO52 has a 30% misprediction rate, the total impact of those mispredicted branches is below 0.5%. A second observation is that while speculating across multiple branches provides some benefits, specially for DYFESM, its cost is certainly not justified. The simplicity of only having one outstanding branch to be resolved is a plus for vector architectures.

All the simulations in the following sections have been performed using a 64-entry BTB and allowing only 1 unresolved branch.

5.4. Reducing the vector queues length

5.4.1. Vector Load Data Queue This section will look at the usage of the vector load data queue. The goal is to determine a queue size that achieves almost the same performance as the 128-slot queue used in the previous sections and yet minimizes as much as possible hardware costs.

Figure 9 presents the distribution of busy slots in the VLDQ 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 VLDQ had a certain number of busy slots For example, for `trfd` at latency 1, the VLDQ was completely empty (zero busy slots) for around 500 millions of cycles.

From fig. 9 we can see that it is not very common to use more than 6 slots. Except for `swm256` and `tomcatv`, 6 slots are enough to cover around 85–90% of all cycles. When latency is increased from 1 cycle to 50 and 100 cycles, the graphs show a shift of the occupation towards higher number of slots. As an example, consider programs `arc2d`, `nasa7` and `su2cor`. For 1 cycle memory latency, these programs have typically 2–3 busy slots. When latency increases, these three programs show an increase in the total usage of the VLDQ, and they typically use around 4–5 slots. 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 execution impact of reducing the VLDQ size can be seen in fig. 10. As expected from the data seen in fig. 9, reducing the queue size to 16 or 8 slots is not noticeable for most programs. Going down to 4 slots affects mostly NASA7 and BDNA but the impact is less than a 1%. Further reducing the VLDQ to 2 slots would start to hurt performance although not very much. The worst case would be again for NASA7 with around a 4% impact. As we have already discussed, 2 slots is clearly a lower bound on the size of the VLDQ to accommodate most memory

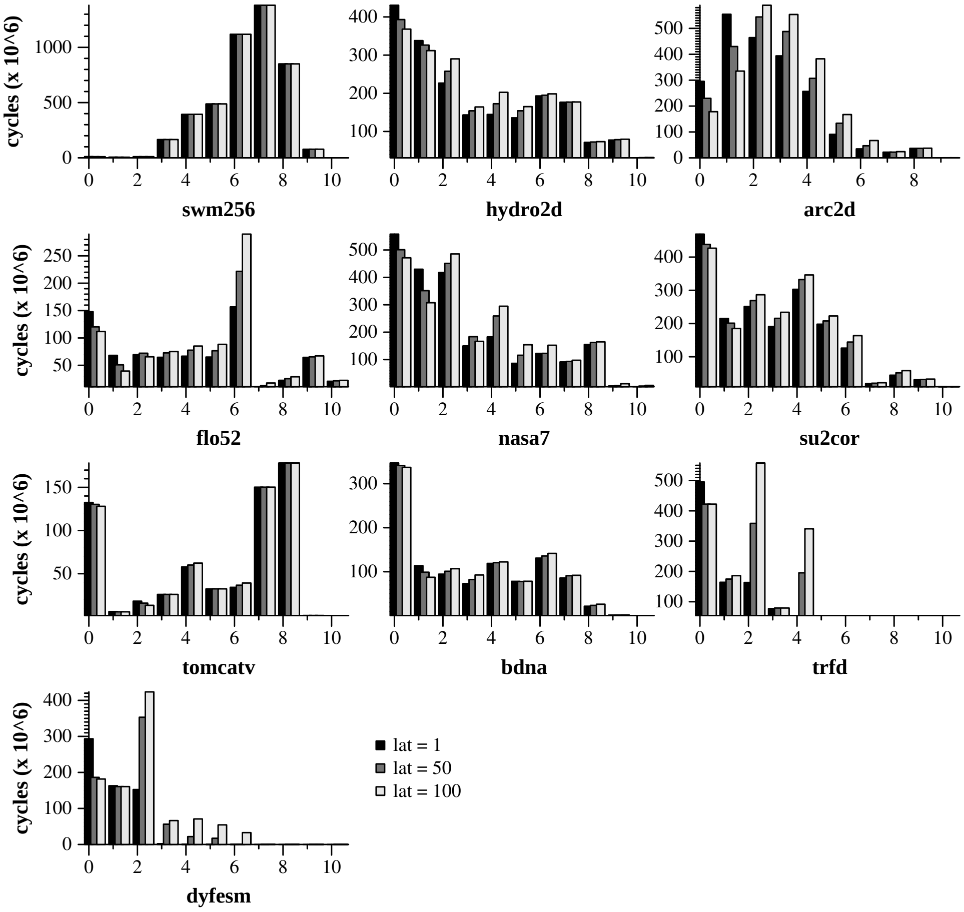

Figure 9. Busy slots in the VLDQ for the benchmark programs for three different memory latency values.

bound loops. Reducing that queue to 1 slot would stop most of the decoupling effect present in the architecture.

Looking at all the data presented in this section we decided to pick a 4 slots VLDQ. All following sections use this size for the VLDQ.

5.4.2. Vector Store Data Queue The usage of the vector store data queue (VSDQ) presents a very different pattern from the VLDQ. Recall that the *AP* always tries to give priority to load operations in front of stores. This has the effect of putting much more pressure on the VSDQ, which can, at some points, become full (even with 128 slots!). This situation is not has unusual as it may seem. As long as the *AP* encounters no dependencies between a load and a store and as long

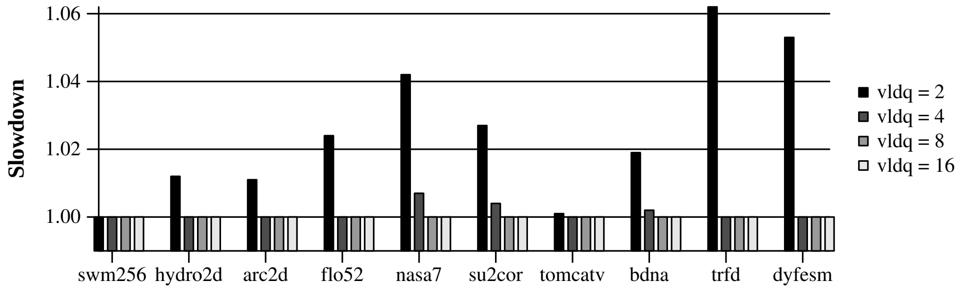

Figure 10. Slowdown due to reducing the VLDQ size (relative to section 5.3).

as there are loads to dispatch, no stores will be retrieved from the VSDQ and sent to memory. Thus the occupation of the VSDQ is much higher than that of the VLDQ.

Figure 11 presents the distribution of busy slots in the VSDQ for the benchmark programs. As we did for the load queue, we plot three distributions corresponding to three different memory latency values. Each bar in the graphs represents the total number of cycles that the VSDQ had a certain number of busy slots. To make the plots more clear, we have pruned some of the graphs. Next to the name of each program we indicate the quartile amount being shown. For example, the full set of data is shown for `nasa7` (q=100%) while the bars in the `hydro2d` graph only present 95% of all available data (the rest of the data set was to small to be seen on the plot). To compensate for this loss of information, each graph also includes the maximum value that the X axis took for that particular program. Again, for `hydro2d`, the graph shows that the maximum occupation of the VSDQ reached 118 slots, although the X-axis of the plot only goes up to 50.

Note that for 6 of the programs, at some point the queue was completely filled (128 full slots), although the most common occupation ranges between 0 and 10 slots.

For the other 4 programs, the occupation of the queue is bounded. For `bdna`, with a maximum occupation of 34 slots in the queue and `su2cor` (maximum 23), the bounding is mostly due to their high percentage of spill code. Each time a vector load tries to recover a vector from the stack previously spilled by a store, the *AP* detects a dependency a needs to update the contents of memory draining the queue. This heavily limits the amount of old stores that are kept in the VSDQ.

Programs `trfd` and `dyfesm` are qualitatively different. These two program simply don't decouple very well. Program `dyfesm` has a recurrence that forces the three main processors, the *AP*, the *SP* and the *VP* to work in lock step, thus typically allowing only a maximum of 1 full slot. Program `trfd` has at its core a triangular matrix decomposition. The order in which the matrix is accessed makes each iteration of the main loop dependent on some of the previous iterations, which causes a lot of load-store dependencies in the queues. These dependencies are resolved, as

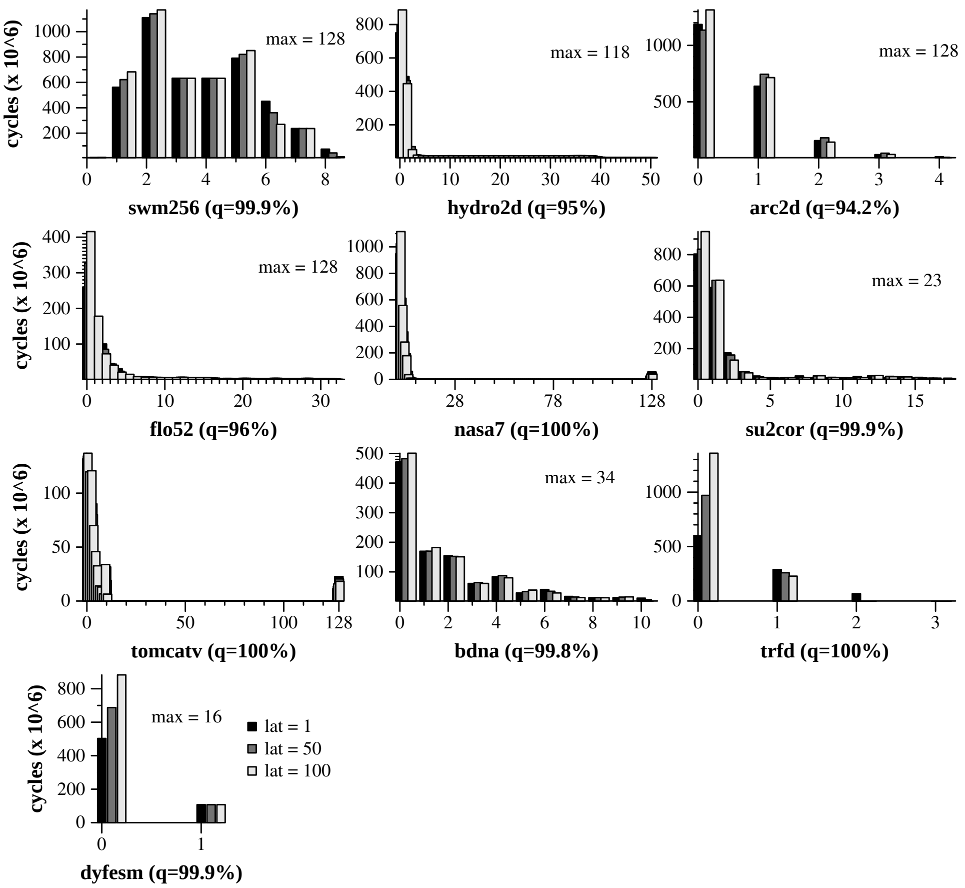

Figure 11. Busy slots in the VSDQ for the benchmarks for three different memory latency values.

in the case for spill code, by draining the queue and updating memory. Thus, the queue does never reach a large occupation.

The execution impact of reducing the VSDQ size can be seen in fig. 12. The bars show that the amount of storage in the VSDQ is not very important to performance. This is mostly due to the fact that we are in a single-memory port environment. No matter how we reorder loads and stores between themselves, every single store will have to be performed anyway. Thus, sending a store to memory at the point where its data is ready or later does not change much the overall computation rate.

From all the data presented in this section we selected the 4-slot VSDQ for all following experiments.

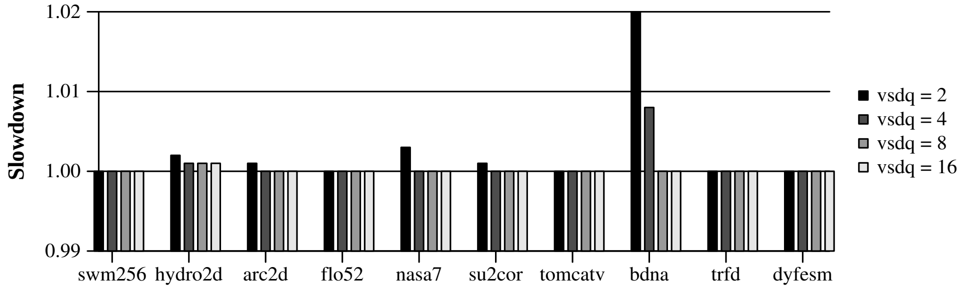

Figure 12. Slowdown due to reducing the VSDQ queues size (relative to section 5.4.1).

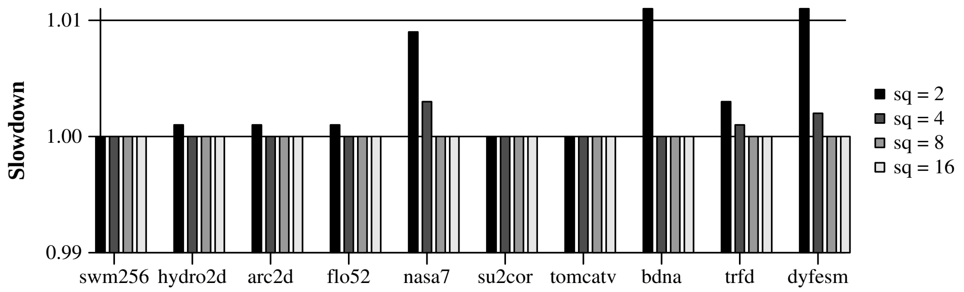

Figure 13. Slowdown due to reducing the Scalar queues size (relative to section 5.4.2).

5.5. *Reducing the scalar queues length*

In this section we will look at the impact of reducing the size of the various scalar queues in the system. Looking back to fig. 2 we will be reducing queues numbered 3–8 and 10–11 from 128 slots down to 16 slots. Queue number 9, the VACQ, *VP*-to-*AP* Control queue, will be reduced from 128 slots to just 1 slot. Note that this queue holds one full vector register used in gather/scatter operations. The 16 slot size is chosen because it is reasonably close to what modern out-of-order superscalar processors have in their queues [31].

The impact of those reductions can be seen in fig. 13. Overall, using an 8 entry queue for all the scalar queues is enough to sustain the same performance as the 128 entry queues. Even for small 2 entry queues, the slowdown is around 1.01 for only 3 programs, `dyfesm`, `bdna` and `nasa7`. Nonetheless, it has to be beard in mind that our programs are heavily vectorized. A small degradation in performance on the scalar side is tempered by the small percentage of scalar code present in our benchmarks. In order to make a safe decision we took a 16 entries queue for all the scalar queues present in the architecture.

Table 2. Latency parameters for the vector and scalar functional units.

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

5.6. Effects of functional unit latencies

In this section we will look at the effects of latencies inside the computation processors of our architecture. So far, all models simulated had all of their functional units using a 1 cycle latency and the vector registers read/write crossbars were modeled as if it was free to go through them. This section will proceed in three steps.

First we will add to our architecture the latencies of the vector functional units. Table 2 shows the values chosen. In a second step, we will add a penalty of 1 cycle of vector startup for each vector operation. In a third step we will add 2 cycles of vector read crossbar latency and then we will add 2 cycles of vector write crossbar latencies. In the last step, we will set the latencies of the scalar units to those also shown in table 2.

Figure 14 shows as a set of stacked bars, the degradation in performance as each of the aforementioned effects is added. The bar at the bottom, labeled "vect. lat" represents the slowdown relative to section 5.5. The following bar, labeled "startup", is the slowdown with respect to the performance of the "vect. lat" bar. Similarly for each of the following bars. Thus, the total height of each bar is the combined slowdown of all these effects.

Figure 14 shows two different behaviors. For seven out of the ten programs, latencies have a very small impact (below 5%). This is due to the fact that decoupling is not only good for tolerating memory latencies but, in general, it helps in covering up the latencies inside the processor. On the other hand, two programs, **`trfd`** and **`dyfesm`** show slowdowns as bad as 1.11 and 1.15, respectively.

The behavior of these two programs is not surprising given what we already saw in section 5.4.1. Both **`trfd`** and **`dyfesm`** have difficulties in decoupling because of the inter-iteration dependences of **`trfd`** and the recurrences in **`dyfesm`**. As we saw, both of them only achieve a very small occupation of the vector load data queue (VLDQ) which indicates a bad degree of decoupling. If we couple this fact with the

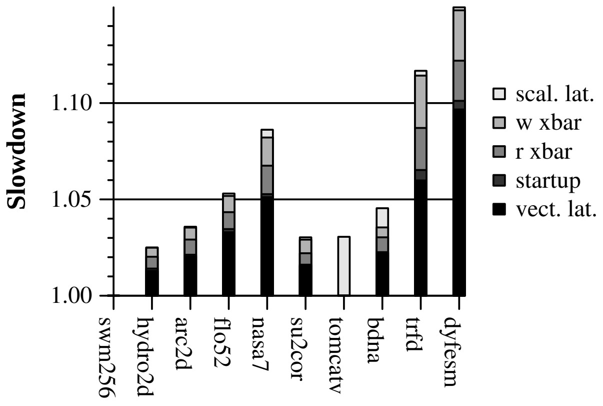

Figure 14. Slowdown due to modeling arithmetic unit latencies and vector pipeline crossbars. (relative to section 5.5).

relatively low vector lengths of `trfd` and `dyfesm`, we see that any cycle added to the vector dependency graph is typically enlarging the critical path of the program.

Going into the detailed breakdown, Fig. 14 shows that the vector functional unit latencies have the highest impact of all latencies added in this section. It is worth noting, though, that the order in which the latencies are added might have some impact on the relative importance of each individual category. Nonetheless, since the vector latencies units are the largest of the latencies added and since the programs are highly vectorized they are the group that most likely will impact performance, as fig. 14 confirms. The startup penalty is only seen in programs `trfd` and `dyfesm`, where its impact is less than a 1%. The vector register file read/write crossbar latencies have an impact on all programs, except, again, `swm256` and `tomcatv`. Typically both latencies have the same amount of impact, being between 0.5 and 1 percentage points for the most vectorized codes and around 2–3 percentage points in the less vectorized `trfd` and `dyfesm`. The scalar latencies have a low impact on all programs, partly due to them being shorter than the vector ones, partly due to the small fraction of scalar code and partly due to scalar latencies masked under other vector latencies.

We decided to compare the impact of functional unit latencies in the reference machine and in the DVA machine. To do so, we simulate a reference machine with no latencies at all and a reference machine with the standard latencies and compute the resulting slowdowns. Then we compare these slowdowns to the slowdowns of Fig. 14, which we just presented. The result of the comparison can be seen in Fig. 15. The results show that in all cases the effect of functional unit latencies is much worse in the in-order reference machine that in the decoupled machine. Since decoupling introduces some form of dynamic scheduling, it can hide latencies that were previously in the critical path by performing memory loads in advance.

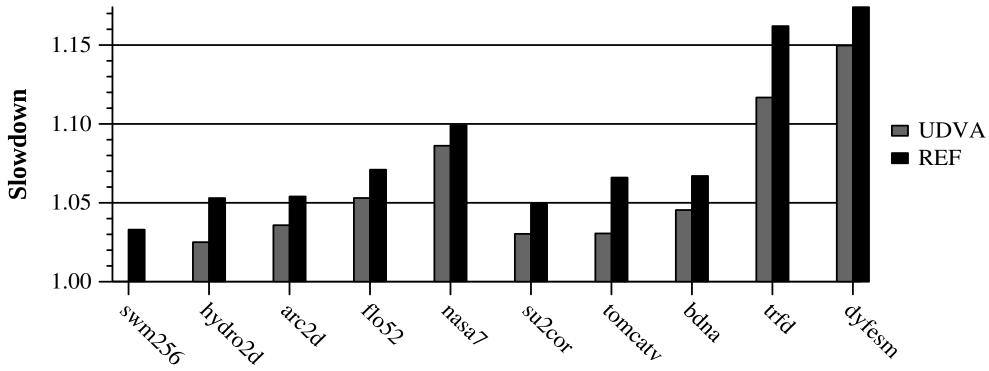

Figure 15. Comparison of functional unit latency impact between the UDVA and REF machines.

5.7. RDVA versus REF

With the data presented in the last section, we have reached a realistic implementation of the originally proposed UDVA. This realistic version will be referred to as RDVA and its main parameters are as follows: all instruction queues and scalar queues are 16 entries long. The address queues in the *AP* are also 16 entries long. The latencies used in the functional units and in the read/write crossbars of the register file are those shown in table 2. The VLDQ and VSDQ each have 4 slots, and the control queue connecting the *VP* and the *AP* has a single slot. The branch prediction mechanism is a 64 entry BTB with at most 1 unresolved branch being supported.

In this section we will re-plot a full-scale comparison of UDVA, RDVA and REF for several latencies. Figure 16 presents the data for the three architectures when memory latency is varied from 1 cycle to 100 cycles. For almost all programs, the difference between the UDVA and RDVA is rather small, and their slopes are relatively parallel. For `swm256`, the difference is almost 0. For programs `hydro2d`, `tomcatv`, `arc2d` and `su2cor`, the slowdown between RDVA and UDVA is less than 1.05 (respectively, it is 1.029, 1.031, 1.037 and 1.044). Programs `flo52`, `bdna` and `nasa7` have a higher slowdown and, moreover, the slope of the curve for the RDVA performance starts diverging from the UDVA at high values of latency. Finally, `dyfesm` and `trfd`, as seen in previous sections, take a significant performance hit when going from the UDVA to the RDVA.

6. Summary and Future Work

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

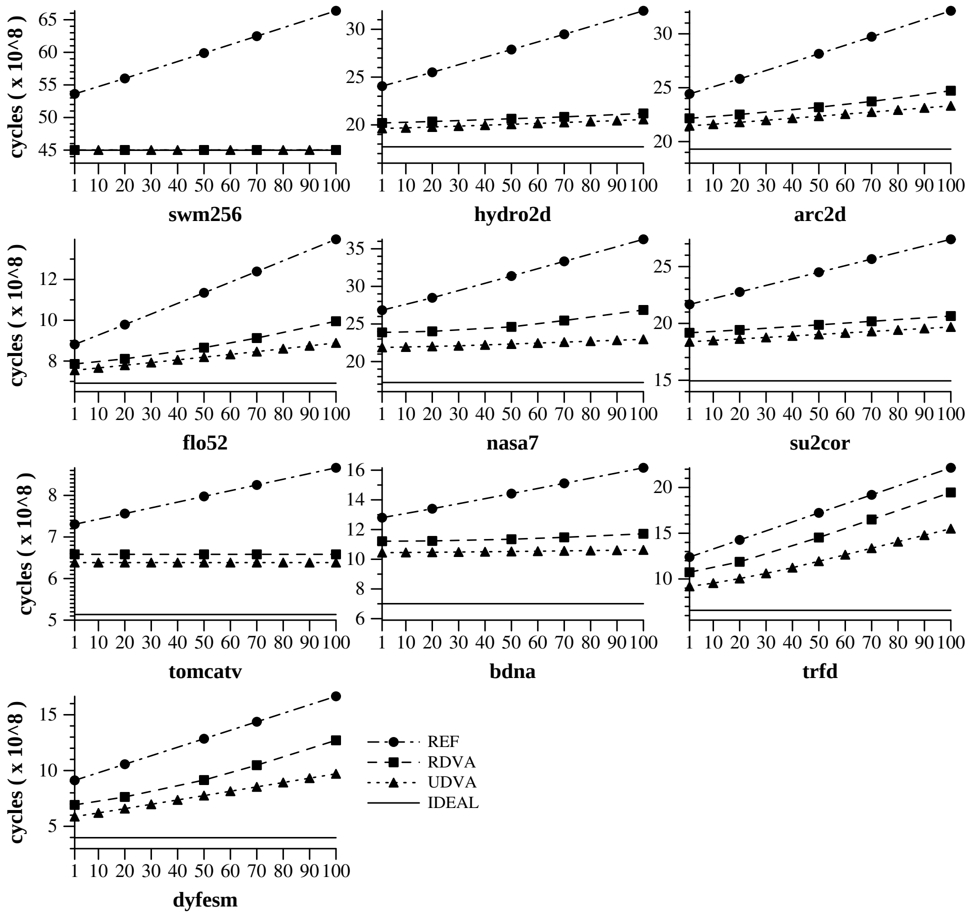

Figure 16. Comparison of REF, UDVA and RDVA execution times for several latencies.

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. When memory latency is set to

50 cycles, for example, speedups of the RDVA over the REF machine are in the range 1.18–1.40, and when latency is increased to 100 cycles, speedups go as high as 1.5.

We have seen that this speed improvements can be implemented with a reasonable cost/performance tradeoff. Section 5.4 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 two elements achieves almost the same performance as one with sixteen slots.

The ability to tolerate very large memory latencies will be critical in future high performance computers. In order to reduce the costs of high performance SRAM vector memory systems, they should be turned into SDRAM based memory systems. This change, unfortunately, can significantly increase memory latency. At this point is where decoupling can come to rescue. As we have shown, up to 100 cycle latencies can be gracefully tolerated with a performance *increase* with respect to a traditional, in-order machine. Moreover, 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.

The simulation results presented in this paper indicate that vector architectures can benefit from many of the techniques currently found in superscalar processors. Here we have applied decoupling, but other alternatives are applying multithreaded techniques to improve the memory port usage [12] and out-of-order execution together with register renaming [13]. Currently we are pursuing the latter approach.

References

1. A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.
2. Bradley J. Benschneider, Andrew J. Black, William J. Bowhill, Sharon M. Britton, Daniel E. Dever, Dale R. Donchin, Robert J. Dupack, Richard M. Fromm, Mary K. Gowan, Paul E. Gronowski, Michael Kantrowitz, Marc E. Lamere, Shekhar Metha, Jeanne E. Meyer, Robert O. Mueller, Andy Olesin, Ronald P. Preston, Donald A. Priore, Sribalan Santhanam, Michael J. Smith, and Gilber M. Wolrich. A 300-MHz 64-b Quad-Issue CMOS RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 30(11):1203–1214, November 1995.
3. M. Berry. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, pages 5–40, Fall 1989.
4. W. C. Brantley and Joseph Weiss. Organization and architecture tradeoffs in FOM. In *IEEE International Workshop on Computer Systems Organization*, March 1983.
5. Tien-Fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching strategies. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 223–232, 1994.
6. E. U. Cohler and J. E. Storer. Functionally parallel architectures for array processors. *Computer*, 14:28–36, September 1981.
7. Convex Press, Richardson, Texas, U.S.A. *CONVEX Architecture Reference Manual (C Series)*, sixth edition, April 1992.

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 Xavier Martorell. Dixie: a trace generation system for the C3480. Technical Report CEPBA-RR-94-08, Universitat Politècnica de Catalunya, 1994.
10. Roger Espasa and Mateo Valero. Instruction level characterization of the Perfect Club programs on a vector computer. In *XV International Conference of the Chilean Computation Society*, pages 198–209. SCCC Press, January 1995.
11. 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.
12. 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.
13. 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.
14. J.R. Goodman, J.T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young. PIPE: A VLSI Decoupled Architecture. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 20–27, June 1985.
15. P.Y.T Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.
16. Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the International Conference on Supercomputing*, pages 323–334, July 1992.
17. L.I. Kontothanassis, R. A. Sugumar, G. J. Faanes, J. E. Smith, and M. L. Scott. Cache performance in vector supercomputers. In *Proceedings of Supercomputing'94*, Washington D.C., November 1994. IEEE Computer Society Press.
18. Lizy Kurian, Paul T. Hulina, and Lee D. Coraor. Memory Latency Effects in Decoupled Architectures. *IEEE Transactions on Computers*, 43(10):1129–1139, October 1994.
19. M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, June 1988.
20. J.K.F. Lee and A.J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.
21. Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 24, October 1996; *Operating Systems Review,* 30(5), December 1996; *SIGPLAN Notices,* 31(9), September 1996.
22. W. Mangione-Smith, S.G. Abraham, and E.S. Davidson. Vector register design for polycyclic vector scheduling. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 154–163, Santa Clara, CA, April 1991.
23. Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.
24. Willi Schönauer and Hartmut Häfner. Explaining the gap between theoretical peak performance and real performance for supercomputer architectures. *Scientific Programming*, 3:157–168, 1994.
25. James E. Smith. Decoupled Access/Execute Computer Architectures. *ACM Transactions on Computer Systems*, 2:289–308, November 1984.
26. 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.
27. James E. Smith, Shlomo Weiss, and Nicholas Y. Pang. A Simulation Study of Decoupled Architecture Computers. *IEEE Transactions on Computers*, C-35(8):692–702, August 1986.
28. Juho Tang, Edward S. Davidson, and Johau Tong. Polycyclic vector scheduling vs. chaining on 1-port vector supercomputers. In *Proceedings of Supercomputing'88*, pages 122–129, Orlando, Florida, November 1988. IEEE Computer Society Press.

29. M. Tremblay, D. Greenley, and K. Normoyle. The design of the microarchitecture of UltraSPARC-I. *Proceeddings of the IEEE*, 83(12):1653–1663, December 1995.
30. Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 392–403, 1995.
31. Keneth C. Yager. The Mips R10000 Superscalar Microprocessor. *IEEE Micro*, pages 28–40, April 1996.