Quantitative Analysis of Vector Code *

Roger Espasa, Mateo Valero, David Padua[†], Marta Jiménez, Eduard Ayguadé

Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya.
c/ Gran Capità, Mòdul D6, 08071 Barcelona, SPAIN
e-mail: roger@ac.upc.es

Abstract

In this paper we present the results of a detailed simulation study of the execution of vector programs on a single processor of a Convex C3480 machine, using a subset of the Perfect Club benchmarks. We are interested in evaluating several cost/performance tradeoffs that the machine designers made in order to asses which features of the architecture severely limit the performance attainable. We present the detailed usage of the vector functional units and a study of the kinds of resource conflicts that stall the machine. The results obtained show that the resources of the vector architecture are not efficiently used mainly due to the single bus memory architecture. Other severe limitations of the machine turn out to be the lack of chaining between vector loads and vector computations, and the lack of a second general purpose functional unit. We also present some data about the port pressure on the vector register file and we see that stalls due to port conflicts are relatively high. We also consider the slowdown introduced by spill code and find that the limited number of vector registers also limits performance.

1 Introduction

In order to design new architectures one has first to properly understand the behavior of current architectures to be able to analyze its strengths and weaknesses and improve future designs. The analysis of the interaction between architectures, compiler technology and application programs is an active field of research where several studies have been carried. This studies try to determine maximum parallelism available in the programs [Jou89, JW89, Wal91, SJH89], frequency of execution of instructions [ML93], bottlenecks and hazards in the architecture [CB94], etc.

In this paper we are interested in the evaluation of a vector architecture [HP90], together with its vectorizing compiler [ZC91]. Since the introduction of the first register-register vector computer, the CRAY-1 [Rus78], compilation technology has evolved to maximize the performance that programs written in high level languages can obtain from a vector architecture. Nevertheless there have been few studies in depth about the relationship between vector architectures, vectorizing compilers and vector programs. In [VH92, VSH91, Vaj91] the CRAY-Y-MP architecture is evaluated through a detailed study of the program characteristics such as number and type of instructions executed, basic block size, fraction of code vectorized, etc. [Lee92] presents a study on the register requirements of vector architectures and analyzes what combination of number of registers and number of read/write ports to the register file has the best cost performance tradeoff.

In this paper we present results obtained from the evaluation of a subset of the Perfect Club [Clu89] programs compiled using the *Convex FC version V8.0* running on a single node

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

[†]University of Illinois, at Urbana-Champaign.

of a Convex C3480 vector machine. To perform this research we have implemented a trace generation tool called dixie able to generate basic block traces from the execution of the programs. This basic block traces are fed into a simulator that we have developed that gives us detailed information, at the cycle level, of every event that happens during the execution of the program. This is in contrast with the tools used in [VH92, VSH91, Vaj91], which only provided statistical data to their authors. It is important to note that we are going to evaluate *automatically* vectorized programs, and thus we will be studying the performance of the architecture together with its compiler.

In section 2 we present the measurement techniques used in this paper. We describe the trace driven approach we have used to simulate the execution of some of the Perfect Club programs. In section 3 we present the benchmarks used in this paper. In section 4 we present the abstract vector machine that we will use to carry the experiments. In section 5 we will study the parallelism exploited by our abstract machine using the output of the Convex compiler. Section 6 will look in detail into the reason that prevent the machine from extracting all the parallelism available and section 7 will study the effects of spill code on vector execution. Finally in section 8 we will present our conclusions.

2 Overview of the measurement technique

The machine on which the experiments were performed is a single processor of a Convex C3480. This machine is ranked in the mini-supercomputer class, and has 8 processors and each one of them has a scalar execution unit and a vector execution unit. We are interested in the vector behavior, and thus all measures have been taken using a single processor running in single-user mode. The C3400 processor can be described as a register-register vector machine. Vector computations can only be performed on data that is in the vector registers and access to memory is through *load/store* instructions. The compiler used in all cases is the *Convex FC version V8.0* with optimization level *-O2* (which implies vectorization). See [Con92] for more details. The vector cpu consists of two functional units. The first one handles all vector operations except multiplication, division and square root. The second one handles all vector operations. Each functional unit has access to 8 vector registers. The vector registers are set up in register pairs, so that each pair has two read ports and one write port. Every vector register is capable of holding 128 elements of 64 bits each. The vector cpu implements fully flexible chaining which means that an operation can be chained to a previous vector operation currently in progress regardless of the cycle at which each operation has started. Due to the variance in response time that the memory system always shows, vector computations can never be chained to *vector load* instructions. Vector *stores* can be chained, though, to vector operations because a set of buffers isolates the vector cpu from the memory latency when sending data to memory.

We have taken a trace-driven approach to gather all the data presented in this paper. We have developed a pixie-like tool called dixie [EM94] that is able to produce a trace of basic blocks executed as well as a trace of the values contained in the *vector length* (`vl`) register. The ability to trace the value of the *vector length* register is critical to have a detailed simulation of the execution of the programs, since each vector instruction can execute with a potentially different vector length. Thus, our measurements do not suffer from the problem reported in [VH92].

Dixie is a tool that given an executable file will produce 1) a modified executable file with instrumentation code that will generate a trace and 2) a basic block description file that maps basic block identifiers to the actual instructions of each basic block. When you run the instrumented executable it will generate a trace of basic block identifiers and a trace of every value that is assigned to the vector length register. This two parallel traces are consumed by a cycle-level simulator that uses the basic block description file to simulate the execution of every single instruction and measure the dynamic behavior of the program. Each time the simulator finds an instruction that loads the `vl` register it will consume a value from the vector length trace. Dixie is able to trace user and library code and, thus, the simulation runs include the user vector code plus all the vector code found in the fortran math libraries. It is important to note that we simulate the output of

	scalar cputime	vector cputime	speedup	vector bb cputime	vector execution %
ADM	105.3	105.1	1.0	67.6	64.3
SPICE	23.2	28.2	0.8	2.4	8.5
QCD	76.4	69.7	1.1	6.4	9.1
MDG	708.6	819.1	0.9	309.4	37.8
TRACK	40.6	39.2	1.0	2.7	6.9
BDNA	203.9	68.4	3.0	42.5	62.1
OCEAN	372.6	312.3	1.2	41.0	13.1
DYFESM	63.9	53.7	1.2	31.0	57.7
MG3D	2310.2	1267.3	1.8	552.6	43.6
ARC2D	800.2	120.6	6.6	118.3	98.1
FLO52	121.1	31.8	3.8	28.9	91.0
TRFD	125.4	71.5	1.8	52.4	73.2
SPEC77	361.5	242.8	1.5	184.9	76.1
TOTAL	5312.9	3229.7	1.6	1440.0	44.6

Table 1: Speedup for the Perfect Club programs on the C3400 machine.

a commercial compiler without introducing any modification in it and that this tracing method gives us absolute precision in all of our measurements.

The simulator we have developed has been based on an abstract version of the Convex architecture and will be described in section 4.

3 Benchmark Programs

We have selected a subset of the Perfect Club programs as our benchmark programs. The Perfect Club application codes are considered to be representative of large typical scientific and engineering programs. For our study, we have executed the thirteen codes on the C3400 in scalar and vector mode and we have obtained the speedups presented in table 3. Column two presents the cpu time (in seconds) of each on of the programs when run in scalar mode. Column three presents the cpu time of the programs (also in seconds) when run in vector mode. Column four is the speedup of the vector versus scalar version. Column five presents an estimation of the fraction of time spent executing vector code. This estimation has been obtained instrumenting all those basic blocks that had some vector code with code that reads the hardware timers (`TTR`) of the Convex machine. Finally, column six presents the percentage of time that this vector basic blocks represent over the execution time of each program. This column can be taken as a rough indicator of the degree of vectorization that each program allows.

As it can be seen from this table, most of the programs do not benefit too much from vector execution, and we believe that for the purposes of our study we should only examine the subset of programs that really exploit the vector cpu. Including programs that have very low speedups in the study would give us no insight into the behavior of the vector cpu, because this programs make very little use of the vector functional units and have very little instruction level parallelism to offer. Thus we have selected the five programs that have greater speedups: *ARC2D*, *FLO52*, *BDNA*, *TRFD* and *SPEC77*. We should really have included program *MG3D* instead of *SPEC77*, but due to the long running time of *MG3D* and the extremely high computation costs of the trace-driven simulator, we have not been able to simulate this program in its full length and thus we have dropped it form the study.

4 The abstract vector machine

The simulator used to gather the performance data for the benchmark programs models an idealized version of the C3400 machine. We feel that in order to better understand vector machines it is important to abstract low level details (like functional unit latencies, technology imposed hardware, restrictions, memory delays and so on) from our study and concentrate in the general behavior of the programs. While the details omitted in the simulator are very important and deserve several studies in its own right, the conclusions obtained from the data gathered with our simulator will still be valid. Since we will be looking at the relative frequency of several different events, the inclusion of the aforementioned low level details would not introduce significant differences in our results.

The architecture studied consists of a scalar part, that we shall refer to as the `SCAL` functional unit, and an independent vector part. The scalar portion is able to execute one instruction per cycle regardless of dependencies, functional unit hazards or branching delays. The vector part consists of two computation units (`FU1` and `FU2`) and one memory accessing unit (the `LD/ST` unit). 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 considered fully pipelined and with a latency of 1 cycle. This asymmetric behavior of the functional units is important when the control unit has to schedule the different operations. Whenever the control unit has to issue an instruction that can be executed both in `FU1` and `FU2`, the decoder will always try to send it first to `FU1` and, if that unit is busy, it will try to send it to `FU2`.

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. 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. This means that there con be no more than two reads and one write simultaneously in progress to the same register bank. The compiler is responsible to schedule the vector instructions and allocate the vector registers so that no port conflicts arise. The machine modeled implements fully flexible chaining [HX88]. Flexible chaining allows for two dependent vector operations to be executed simultaneously without imposing restrictions in the issue time of the two instructions. Older vector designs, like the CRAY-1, had a fixed chaining scheme in which chaining could only occur if the second operation of a dependent pair was issued at a particular point in time (more precisely, it occurred if the second operation was issued a certain number of cycles after the first operation). The chaining implementation we have chosen to model has two read and one write *pointers* for each one of the vector registers. This read/write pointers control the next element that has to be sent to the functional units and allow that the same physical vector register can be shared by different instructions that have started at different cycles.

The `LD/ST` unit can only service one request to/from memory at time, because the architecture simulated has only one bus connecting the cpu to memory. The memory system simulated is an ideal one that has a 1 cycle latency and delivers one datum per cycle, regardless of the stride used. The real C3400 architecture has one additional limitation regarding the memory system that we have chosen to simulate. Despite the fact the memory delivers one datum per cycle, we will not allow to chain the result of a vector load instruction with a vector computation instruction. This limitation is a common problem that multiprocessor vector architectures have to face because the variance in response time of real memory systems makes it difficult to predict the arrival time of a datum to the processor. Thus, while it is not impossible to chain vector computations to vector loads, a reasonable tradeoff is to restrict this chaining. We believe it is important to simulate this feature of the Convex C3400 architecture in order to evaluate its impact on performance.

Row	Functional units in use				% of execution cycles			
	FU2	FU1	L/S	SCAL	weighted mean	arithmetic mean	(w. mean) (collapsing)	(a. mean) (collapsing)
0.	0	0	0	0	0.00	0.00		
1.	0	0	0	1	11.45	11.00	11.45	11.00
2.	0	0	1	0	34.98	37.00		
3.	0	0	1	1	1.00	0.80	35.98	37.80
4.	0	1	0	0	0.46	0.00		
5.	0	1	0	1	0.15	0.00	0.61	0.20
6.	0	1	1	0	4.80	5.80		
7.	0	1	1	1	0.38	0.00	5.18	6.20
8.	1	0	0	0	5.43	4.60		
9.	1	0	0	1	0.24	0.00	5.67	4.60
10.	1	0	1	0	13.13	11.60		
11.	1	0	1	1	0.83	0.20	13.96	12.20
12.	1	1	0	0	6.72	5.80		
13.	1	1	0	1	1.41	0.80	8.13	7.00
14.	1	1	1	0	16.47	14.00		
15.	1	1	1	1	2.54	2.80	18.01	17.60

Table 2: Utilization of the vector functional units.

5 Instruction Level Parallelism

We run each one of the five programs from the Perfect Club and simulate its execution cycle by cycle. The simulator reads a trace of basic block addresses and executes each instruction in the basic block following the issue rules stated in section 4. At every single cycle we keep track of how many functional units are simultaneously busy. This number gives us an idea of the amount of parallelism present in the program that we are actually exploiting. Note that in our simulated architecture the maximum parallelism achievable (ignoring the scalar unit) is 3. In this section we will look into the parallelism that the architecture is able to exploit, and in the following sections we will consider what are the factors that limit this parallelism.

Table 2 presents the utilization of the vector units. Each row in the table presents a different "state" of the vector functional units. A value of 1 in the columns two to five indicates that the corresponding unit is active. Thus, row number 0, with code 0000, corresponds to the machine being idle, and row number 1, with code 0001, corresponds to pure scalar execution. The last row, with code 1111, corresponds to maximum efficiency: all three vector units (FU1, FU2 and LD/ST) and the scalar unit are working simultaneously. Columns six to nine present the fraction of cycles that the machine was in each one of the states. Columns six and seven present the fraction of cycles that the machine was in each one of the states, using the weighted mean (each program contributes to the mean proportionately to its running time) and the arithmetic mean. Columns eight and nine are obtained when considering only the vector functional units and ignoring the state of the scalar unit. Notice how every two rows are exact in its vector portion and only differ in the activity of the scalar unit. We have "collapsed" every two rows by adding them and we have the results presented in columns eight (the result of collapsing the sixth column) and nine (the result of collapsing column seven).

Row number 1 in table 2 corresponds to pure scalar execution. Even for our benchmark programs that are highly vectorizable there will always be some portion of scalar code mostly related to library code for input/output, scalar code generated to set up the environment for vector computation and code corresponding to portions of the program that current vectorization technology can not handle.

Row 2 represents the situation where the vector memory unit is the only functional unit working, while in row 3 we have the percentage of cycles that the load/store vector unit and the scalar unit have been running simultaneously. If we look either at column 8 or 9 we can see that the fraction of cycles spent in this two states is extremely high. Let's assume that scalar code is not useful for the computation, in the sense that scalar code is just overhead code to compute addresses, perform calls, jumps, control loops, etc. Table 2 shows us that in 35.98% of the cycles we are *not* producing any results. We are either moving data or doing "setup" work. This high number of unproductive cycles is due to several reasons. First, all applications have initialization loops that just initialize the data structures to be used during the program which only have vector memory operations. Second, the architecture only has one memory bus, so whenever the instruction issue stage finds two consecutive vector memory operations in the code, it will stall waiting for the first memory operation to complete. At best, the decoder will be able to issue a few scalar instructions found between the two memory operations, but this only happens in a few number of cycles (1%). Third, the architectural limitation of not being able to chain vector computation instructions to vector loads is also responsible for stalling the machine to wait for a memory operation. In the next section we will quantify each one of these effects.

Rows 4 and 5 present rather unusual situations. In these rows we have that the only vector unit working is `FU1` and the scalar unit can be working (row 5) or not (row 4). This can happen whenever a) there is no parallelism in the code, that is, we have a single vector computation isolated between a long scalar section of code or b) whenever there has been a port conflict between the vector instruction and its sequential follower. Both cases are rather unusual in the programs studied, as we can see by the low (0.61%) fraction of cycles that they represent.

In rows 6 and 7 we see some degree of overlapping between computation and memory accessing instructions. We have that both the restricted vector functional unit and the load/store vector unit are working concurrently. the typical sequences of code that put the machine in these two states are sequences where we have issued an instruction to functional unit 1 and a memory instruction to the `LD/ST` unit (they can be related or unrelated) and we encounter in the instruction stream a) a computation instruction that might be dependent on the memory instruction in which case a "load chain" conflict arises and we have to stall the machine, or b) we find a second memory instruction that will have to wait until the one running completes or c) there are simply no more vector instructions to be issued, which happens at the end of loops, for example.

Rows 8 and 9 are similar but not equivalent to rows 4 and 5. They represent situations where the only vector functional unit working is the general purpose functional unit. But the reasons that leads us to states 8/9 or states 4/5 are rather different. In rows 4/5 we were talking about lack of parallelism or port conflicts. Rows 8/9, as we will see more in depth in the next section, can also be the result of a port conflict but more usually they will be the result of the *presence* of parallelism. If we have to consecutive vector instructions that *require* the `FU2` unit (for example, any mix of consecutive multiplications, divisions and square root will do) the second instruction will have to stall waiting for the first one to free the functional unit. This situation is rather frequent, and corresponds to compute bound loops that have a lot of vector instructions that can only execute in `FU2`. Note the difference of cycles percentage between rows 8 and 9 (5.67%) and rows 4 and 5 (0.61%). The decision to put a second functional unit that can only perform a subset of all operations instead of having a general purpose one has a significant negative impact on performance. Next section will provide more data to discuss this tradeoff in depth.

Rows 10 and 11 represent the overlapping of `FU2` instructions with vector memory operations. Again, this two rows have a higher percentage of cycles (13.96) than rows 6 and 7, mostly because of the same reasons explained for rows 8 and 9. See how the four rows together represent almost a 20% of all executed cycles. It is very important to remark that our simulator treats all vector operations as being fully pipelined. Had we decided to take into account the real latencies of instructions like division or square root (that could be well beyond 10 cycles witch current technology) we would see how the number of cycles and percentage that rows 8 through 11 represent would be much higher. For example, if we chose a 10 cycle latency for division, the execution of a vector division of a given vector length would take as much time as 10 vector additions of the same vector length. This means that the consequences of having a second functional unit that cannot perform certain frequent operations would actually be worse in a real machine than what we have found in this paper. For the sake of simplicity, we have chosen the 1 cycle latency approach to highlight the relative importance of the architectural decisions involved in a ideal vector architecture but without getting into implementation issues.

The last four rows represent the states where the maximum parallelism is achieved by the architecture. In all of them, both vector computation units are working concurrently and producing *two* results per cycle. Nevertheless, this peak efficiency is only reached in 26.14% of all executed cycles. The rest of the cycles are divided between a) approximately 50% of all cycles the vector computation units are idle, that is, we are executing scalar code or just moving data around and b) in 26.42% of the cycles only one of the functional units is producing some result. If, again, we assume that scalar code is not useful for the computation because it's just overhead code to compute addresses, perform calls, jumps, control loops, etc., and we also assume that vector loads and store are *not* part of the computation, we have a very low average number of results computed per cycle. We have that in 26.14% of cycles we are producing 2 results, in 26.42% of cycles we are producing just 1 result and in 47.43% of the cycles we are producing 0 results. This gives us an average of 0.74 results per cycle. Even taking into account that this is a lower bound and that, in fact, there is also scalar code that can not be considered "overhead" since it is actually producing results, it is still rather far from the peak performance of 2 results per cycle.

Looking at table 2 globally, we can note some other interesting points. First, the fraction of code that is hidden by vector operations is rather high. Adding the seven rows where the scalar unit is active and at least one of the vector units is active we see that this overlapping happens in 6.55% of all cycles. The percentage of cycles where we are executing scalar code without overlapping is 11.45%. As in our simulator one cycle is equivalent to one scalar instruction, we can see that 56.3% of all scalar instructions have been hidden by vector instructions. For the rest of instructions not overlapped, we believe that modern vector processors should have a superscalar processor to execute the scalar portion of programs, and thus the 11.45% of cycles could be further reduced.

Second, the total number of cycles where the vector `LD/ST` functional unit is accessing memory is very high. Since we assume a perfect memory system, we can consider that in every single cycle that this functional unit was busy, a datum was being transferred from/to memory. Adding all rows where the `LD/ST` unit is busy, we get that in a 73.13% of all executed cycles we were accessing memory. On the other hand, we have already seen how in 79.7% of all cycles a result was produced. The comparison between this two numbers gives us insight into the balancing of the programs studied. They turn out to be slightly compute bound in terms of total number of "abstract" operations performed. Nevertheless, we have to insist in the fact that neither memory latencies nor functional unit latencies have been taken into account and these are two very important factors that could change this balancing. This is a subject that deserves further investigation.

Finally, we need to better understand what are the factors that limit the machine and prevent it from achieving full efficiency. We have argued that some of this factors could lie in the single bus architecture, the load chain problem or the asymmetry between the two functional units. In the next section we will quantify the relative importance of all this factors.

6 Limitations to instruction level parallelism

Our architecture introduces several resource limitations that can be classified into: a) not enough functional units, b) the lack of the ability to chain a computation to a load instruction and c) conflicts in the ports of the vector register file. There is still another indirect limitation introduced by the limited number of vector registers. It is d) the spill code instructions used to move data between registers and memory when no free registers are available to perform a certain computation. In this section we will deal with limitations a), b) and c) and in the next section we will look into the spill code problem.

The simulator is able to collect statistics about all the hazards that occur during the execution of the programs. When an instruction is not issued we determine all the reasons that prevented the instruction from executing and store in a table the total number of cycles that the decoder had to stall due to those reasons. For example, in the following code:

```
1. add  v0,v1,v2
2. mul  v2,v0,v3
3. add  v0,v3,v1
```

we see how the third instruction can not be issued for two reasons. First, it has a read port conflict with the two previous instructions. Instructions 1 and 2 use the two read ports available in register bank 0. Thus, no other instruction can simultaneously access any of the registers in that bank. In particular, instruction 3 has a read port conflict in its first operand (`v0`). Second, the two computational units in the vector cpu are busy and the third instruction will have to wait until one of the previous instructions finishes its execution and releases its functional unit. When this situation arises, the simulator stores in a table the total number of cycles that the third instruction had to wait prior to execution due to this combination of hazards.

As another example, consider the following two instructions:

```
1. ld  effa1,v0
2. st  v0,effa2
```

This piece of code is just moving data in memory. The second instruction stalls the decoder because it has two conflicts with the previous one. First, our machine only has one load/store unit, so it's not possible to start the store in parallel. Second, even if we had more busses connecting the cpu and memory we have decided that we could not chain instructions to the result of a load, so the store instruction would have to wait until the load completed. We term this latter situation as a "load chain conflict".

The effects of limitations a), b) and c) can be seen in table 3. Each row in the table corresponds to a different situation that stalled the machine. To understand this table, consider all vector computation instructions divided into two classes: a FU2-class instruction is an instruction that can *only* execute in the `FU2` functional unit. That is, we have three FU2-class instructions: `mul`, `div` and `sqrt`. The rest of vector computation instructions are FU1-class instructions because they can execute in *both* functional units. The first five columns in table 3 have the following meanings: In column labeled `FU2`, a 1 indicates that the machine was stalled because there was no functional unit available to execute a FU2-class instruction. In column labeled `FU1`, a 1 indicates that the machine was stalled because there was no functional unit available to execute a FU1-class instruction, which means that *both* functional units were busy. In column labeled `LD`, a 1 indicates that the machine was stalled because the vector `load/store` unit was busy and could not accept more memory instructions. In column labeled `PRT` a 1 indicates that a certain instruction was not issued due to conflicts in the read/write ports of the vector register banks. Finally, a 1 in column labeled `LDX` (*load chain*) indicates that the machine was stalled because a computation was dependent on a load instruction and, thus, could not be chained to it. The next two columns show the percentage that each one of these hazard combinations represent over the total number of executed cycles (using the weighted and arithmetics means).

Row 0 presents the percentage of cycles that the decoder stalled due to the inability to chain a computation to a load. Having a 1 only in column `LDX` means that we were ready to issue a computation instruction, we had a free functional unit, we had ports to access the vector registers needed as operands, but one of those operands was the result of a previous load still in progress. The percentage of cycles lost in this situation is very high (16.42%). If we add together all rows where `LDX` is 1, we see that the lack of chaining with loads is co-responsible for as much as 25% of stall cycles. Other vector machines, like the Cray-YMP C90 do not have this chaining restriction and can better utilize the functional units. Alternatively, if the machine had more registers, these cycles could probably be filled with useful computations by unrolling the loops and scheduling the operations such that vector loads and dependent computations could be moved apart as much as possible. Anyway, bear in mind that except for row 0, in all other rows the `LDX` problem is not the only cause that stalls the machine. Thus, if we removed the "load chain" restriction we would be sure of eliminating those 16.42% of cycles accounted in row 0, but probably not many more.

	Hazard combination					% of execution cycles	
						weighted	arithmetic
Row	FU2	FU1	LD	PRT	LDX	mean	mean
0.	0	0	0	0	1	16.42	18.01
1.	0	0	0	1	0	1.62	1.57
2.	0	0	0	1	1	2.59	3.19
3.	0	0	1	0	0	30.06	31.37
4.	0	0	1	0	1	0.65	0.61
5.	0	0	1	1	0	3.18	3.30
6.	0	0	1	1	1	0.01	0.01
7.	0	1	0	0	0	2.94	3.14
8.	0	1	0	0	1	1.28	1.06
9.	0	1	0	1	0	2.11	1.82
10.	0	1	0	1	1	0.36	0.29
11.	1	0	0	0	0	8.29	7.02
12.	1	0	0	0	1	2.18	1.70
13.	1	0	0	1	0	3.48	2.98
14.	1	0	0	1	1	1.51	1.18

Table 3: Relative importance of the hazards occurred during the execution of the six benchmark programs.

Row 1 presents the number of lost cycles due exclusively to port contention. Even though this row does not represent a very large fraction of all executed cycles, if we add together all rows where column `PRT` has a 1, we see that there is a high number of wasted cycles (14.86%) due to port contention. The architecture provides 8 read ports and 4 write ports to the vector registers and the maximum possible simultaneous requests are 4 reads and 2 writes for the functional units and one additional read or write for the `LD/ST` unit, but still this seems not to be enough to sustain the bandwith required by the vector functional units. We have looked in detail at this port conflicts and found that the overwhelming majority of conflicts are due to reads. In particular, in 15.02% of all executed cycles the machine was stalled due to a conflict in one or two of the read ports that the candidate instruction for being issued required. In contrast, only 1.19% of all executed cycles the reason of a stall was a write port. Note that these two percentages are not independent, because there are instructions that cause both read and write port conflicts simultaneously.

Row 3 is the one that accounts for the largest percentage of stall cycles (30.6%). It represents the hazards occurred when the decoder tries to issue two consecutive vector memory instructions. This is a very common situation in vector loops for several reasons. If the compiler does not perform software pipelining and/or loop unrolling, the typical starting code of a loop is a sequence of loads that will bring the data to be operated on. This sequence of loads will always conflict in our architecture. Also, at the end of the loop there is usually a sequence of several stores that will conflict between them and will also conflict with the load instructions at the beginning of the next iteration of the loop. Note that in this row the only conflict is the memory unit, thus increasing the number of busses to memory should decrease the percentage of cycles stalled due to this reason. If we also add all rows where `LD` is 1, we have that the lack of more memory units is co-responsible for stalling the machine in 33.9% of all executed cycles.

Row 4 presents a very special case. Whenever we have a load followed by a dependent store, we have two conflicts. First, we have a 1 in column LD because we need the load/store unit to issue the store but it's busy servicing the load. Second, even if we had a second memory unit, the load chain restriction would prevent us from issuing the store. Thus, we also have a 1 in column LDX. Note that this situation is fairly rare, and corresponds mostly to loops that do initialization work, like copying arrays or initializing memory. Row 6 correspond to a still more special case in which the store following the load could also not be issued due to the fact that the read ports of the corresponding register bank were both busy (most probably, some other previous instruction is using them).

Row 5 is a more common situation. Consider the following code:

```
1. add v2,v3,v0
2. st  v0,effa
3. ld  effa2,v0
```

This code can easily arise in loops. The result of the addition is stored in some array but is no longer needed in the rest of the loop body. After instruction 2, the compiler knows that register `v0` is dead, and reuses it to bring some other value from memory. When trying to issue instruction number 3, we have a `LD/ST` functional unit conflict and also a write port conflict to register `v0`, because the only write port to the register bank where `v0` belongs is busy servicing instruction number 1. Obviously, there are other situations where this combination of conflicts can arise.

Let's consider rows 7 through 10. When we have a 1 in column `FU1`, its meaning is that *both* functional units were busy and so we could not issue a FU1-type instruction. This four rows are showing us that the programs had some more parallelism available that could be exploited by adding more functional units. If we ignore for the moment that rows 8, 9 and 10 present port/load chain conflicts, the lost cycles that could have been successfully used with a third functional unit is *at least* a 6.69% of all executed cycles. Actually, if a third functional unit was added we could probably reduce more cycles of execution, but this will remain a topic for future research for the moment.

As we have also noted in the previous sections, the pressure on the `FU2` unit is very high due to the fact that `FU1` is not able to execute multiplications, divisions and square roots. This can easily be seen in rows 11 to 14, where the need for another functional unit of type FU2 stalls the machine in 15.46% of all executed cycles. It is also true that rows 12 and 14 have also the load chain conflict but they represent only a 3.69% of the executed cycles.

7 Spill code

When the compiler is allocating registers in a basic block file it may find itself without any free register in which to store a result. In this situation, the compiler has to insert special instructions, called *spill code* that will save a certain register to memory (spill it), to be able to reuse that register to store some other value. The contents of the register spilled to memory will be reloaded at some later point in time to some other free register. This spill load/store instructions are not part of the computation but are an overhead introduced by the compiler due to the limited size of the register file. However, spill code is a negative contribution to performance only if it actually increases the minimum number of chimes required to execute ine iteration of the loop where it appears. If a loop with spill is highly compute bound, it means that there will most probably be many cycles where the `LD/ST` unit will be free to be used by the spill instructions, and they will effectively be hidden by the computation instructions. On the other hand, if a loop is memory bound, every single spill instruction will lengthen its initiation interval at least by `vl` cycles, where `vl` will depend on the length of the vector registers.

If we had an architecture with infinite registers there would be no spill code at all. In this section we will study the effect of spill code on performance. There has been some previous work on the effects of spill code in vector processors. In [Lee92] it is suggested that the effect of spill code is not very bad in vector processors because spill can be hidden in those cycles where the load/store unit is not busy. Nevertheless, in [Lee92] the architecture under study was a CRAY-Y-MP which has the same number of functional units as our architecture but has 3 paths to memory (two load busses and one store bus) and thus has many more opportunities to hide those spill instructions. We believe that in our single bus architecture the effect of spill code will be significantly higher.

To study the effects of spill on performance we have modified the simulator so that each time it finds a vector spill instruction it ignores it. This way we are creating the effect of having an infinite register file, and we are eliminating the negative effect of spill code. We run the programs

	original cycles	nospill speedup
BDNA	1053.7	1.49
ARC2D	2236.8	1.07
FLO52	726.3	1.07
TRFD	875.4	1
SPEC77	2472.1	1.01

Table 4: Speedup obtained when eliminating all the spill code from the programs.

again with this version of the simulator and we obtain the total number of cycles needed to execute them. We can see the effect of spill code by comparing this results with the results obtained with the original simulator.

Table 4 presents the results. First column in table 4 is the total number of cycles (in millions) needed to execute the original program and the second column is the speedup obtained when eliminating all the spill from the programs. We can see the great variance in speedup between the different programs. `BDNA` happens to have extremely long basic blocks, coming from very long loops, and the compiler has to introduce a lot of spill code to execute them with only eight registers. We have measured the mean size of BDNA vector basic blocks and it's close to 120 instructions (not including the scalar instructions). From this table we can conclude that in a single bus architecture spill code has a negative effect on performance, especially for those programs that have a high register pressure.

8 Conclusions and Future Work

In this paper we have presented quantitative measurements of the execution of vector code produced by a commercial compiler on a vector supercomputer. We have chosen a subset of the Perfect Club benchmarks and executed it using a simulator. Results show that the fraction of time executing vector computations is not as high as one would expect (around 50%). We have presented data about the fraction of utilization of the vector functional units and found that only in roughly 5% of the cycles are all vector computation units busy, while there is a 25% percent of cycles where only one of the two arithmetic units is working and the last 50% of the cycles are either load/store cycles or purely scalar cycles. The major limitation that prevents full utilization of the machine is the single bus to memory architecture, which even in the case of adding infinite number of functional units and ports, would be responsible for stalling the machine in 33.9% of all executed cycles. Related with the memory problems is the inability to chain loads and computation. This restriction is responsible for stalling the machine in a combined 25% of all executed cycles. Additional limitations found were the lack of a second functional unit able to perform multiplication and division (15.46% of executed cycles) and limited number of ports to access the vector register file (14.86%). We have also seen that vector spill code is not negligible and that it produces an average slowdown of 8%.

The tools we have developed for this study are currently being used to evaluate solutions to the problems reported in this paper. We are currently looking at different alternatives to solve the spill problem, as well as reducing the number of conflicts in the vector register file ports. We are also investigating new functional unit schemes to reduce the number of lost cycles.

References

[CB94] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of APLHA AXP performance using TP and SPEC workloads. *International Symposium on Computer Architecture*, pages 60–70, 1994.

[Clu89] Perfect Club. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, pages 5–40, Fall 1989.

[Con92] Convex Press, Richardson, Texas, U.S.A. *CONVEX Architecture Reference Manual (C Series)*, sixth edition, April 1992.

[EM94] Roger Espasa and Xavier Martorell. Dixie: a trace generation system for the C3480. Technical Report CEPBA-RR-94-08, Universitat Politècnica de Catalunya, 1994.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[HX88] Kai Hwang and Zhiwei Xu. Multipipeline networking for compound vector processing. *IEEE Transactions on Computers*, 37(1), January 1988.

[Jou89] Norman P. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, 38(12):1645–1658, 1989.

[JW89] Norman P. Jouppi and David W. Wall. Available instruction level parallelism for superscalar and superpipelined machines. *ASPLOS*, pages 272–282, 1989.

[Lee92] Corinna G. Lee. *Code Optimizers and Register Organizations for Vector Architectures*. PhD thesis, University of California at Berkeley, 1992.

[ML93] Larry McMahan and Ruby Lee. Pathlengths of SPEC benchmarks for PA-RISC, MIPS, and SPARC. *COMPCON*, 1993.

[Rus78] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[SJH89] Michael D. Smith, Mike Johnson, and Mark A. Horowtiz. Limits on multiple instruction issue. *ASPLOS*, pages 290–302, 1989.

[Vaj91] Sriram Vajapeyam. *Instruction-Level Characterization of the Cray Y-MP processor*. PhD thesis, University of Wisconsin, Madison, 1991.

[VH92] Sriram Vajapeyam and Wei-Chung Hsu. On the instruction-level characteristics of scalar code in highly-vectorized scientific applications. *IEEE Micro 25*, pages 20–28, 1992.

[VSH91] Sriram Vajapeyam, Gurindar S. Sohi, and Wei-Chung Hsu. An empirical study of the Cray Y-MP processor using the PERFECT Club benchmarks. *International Symposium on Computer Architecture*, pages 170–179, 1991.

[Wal91] David W. Wall. Limits of instruction level parallelism. *ASPLOS*, pages 176–188, 1991.

[ZC91] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers*. ACM Press, New York, NY, 1991.