This is a two-column version of the paper appeared in *Supercomputation in Nonlinear and Disordered Systems: Algorithms, Applications and Architectures* , World Scientific Publishing, 1997.

On the Instruction level characteristics of the SPECfp92 programs on a vector computer

Roger Espasa Mateo Valero *

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

Abstract

In this paper we study the instruction level characteristics of the SPECfp92 programs when compiled and executed on a vector processor. Using a trace driven approach we measure the degree of vectorization of the programs, the vector length used in operations, the operation type distribution and the balance between memory and compute operations. We also study the spill code introduced in the program by the compiler and the pressure on the dispatch unit of the vector architecture.

1 Introduction

Vector architectures have historically been evaluated using high level benchmarks. By high level benchmarks we understand measuring the performance of a vector machine using benchmarks like the Perfect Club [1] or even LINPACK [2], that summarize the behavior of the machine in the total MFLOPS achieved. While these procedures certainly give some interesting information to the final user of the machine, they can not capture the performance advantages/disadvantages that each different vector architecture offers. We believe that a better approach to understand and characterize the behavior of a vector machine is the measurement of the low level (instruction level) activity of a collection of benchmarks.

While these approach has been routinely applied by the microprocessor community, there are very few studies of this kind in the vector computer arena. Microprocessor design is more and more driven by performance studies that try to identify all kinds of parallelism available in the instruction stream [3, 4, 5], and that evaluate possible bottlenecks in the architecture. The detailed knowledge of the relative frequency of execution of instructions [6, 7] leads to the inclusion of new compound operations (like the fused multiply add instruction) in the instruction set so that the "common case" is optimized and executes in fewer cycles. On the other hand, vector machines tend to offer more power by moving into the multiprocessor world and by reducing cycle time, but the basic architecture of the vector processor has not changed very much since the introduction of the Cray-1 [8]. Moreover, as CMOS technology continues advancing, the cycle time difference between expensive ECL vector supercomputers and low-cost microprocessors is getting smaller and smaller. Thus, it won't be possible to sustain the performance advantage of vector computers solely on the cycle time. Future vector architectures will have to make as much use as possible of the instruction level parallelism found in scientific programs. New designs will have to borrow as many techniques as possible from the superscalar world, and mixed designs with an off-the-shelf superscalar microprocessor tightly coupled to a vector cpu will become more common. Vector cpus will have to deal with out-of-order execution, schemes to reduce the register pressure

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

and an increase in the number of computational functional units. In order to evaluate the tradeoffs involved in all these architecture features, the instruction-level behavior of programs has to be measured. In [9, 10] the Cray-YMP [11] architecture is evaluated through a study of the scalar and vector instruction level characteristics. In [12, 13] detailed measuring models of vector architectures are presented. In [14] a study of the Cray-YMP vector register usage was carried out and several alternative vector register configurations were proposed. Nevertheless, there are few studies that measure the relative importance of vector instructions, the frequency of stalls, or the instruction level parallelism.

This paper will present data on the instruction level characteristics of the SPECfp92 programs when compiled using a production vector compiler. We have implemented a tracing tool called Dixie [15] able to generate a detailed instruction level trace of all the program execution (including user plus library code). With this tracer we are able to measure at a fine level the degree of vectorization of the programs, the vector length used in operations, the operation type distribution and the balance between memory and compute operations.

The structure of this paper is as follows. Next section will present our measurement technique. Section 3 and 4 will present the basic operation counts for the benchmarks. Section 5 will present the vector length distribution of the programs. Section 6 will look at the spill code generated by the compiler. Section 7 will present measurements on the pressure on the dispatch unit. Finally, in section 8 we will present our conclusions.

2 Overview of the measurement technique

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 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 program execution, since each vector instruction can execute with a potentially different vector length. Thus, our measurements do not suffer from the problems reported in [10].

Dixie has been targeted to a Convex C3480. This machine is a vector multiprocessor with 8 processors. Each individual processor has a scalar and a vector unit. The vector unit consists of two functional units (one is fully general purpose and the other only performs add-like operations) and one memory accessing unit. All these functional units are connected to a single vector register file which is organized in banks. It has 4 banks which hold 2 vector registers each. The vector registers hold 128 elements of 64 bits. Each bank has 2 read ports and 1 write port. The machine implements fully flexible chaining [16] except for loads, which can not be chained to a computation instruction. See [17] for further details. We have compiled all the *SPECfp92* benchmarks using the fortran compiler (*Convex FC version V8.0*) with optimization level *-O2* (which implies vectorization). Then, the output of these compilations are fed into Dixie which produces 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 generates 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 a commercial compiler without introducing any modification in it and that this tracing method gives a high precision in all of our measurements.

3 Basic Operation Counts

Table 1 presents the basic operation counts for the eleven programs of the SPECfp92. First column in this table presents the total number of basic blocks (in millions) executed for each program. Next two columns present the total number of instructions issued by the dispatch unit, broken down into scalar and vector instructions. In this paper we will distinguish between *instructions* and *operations*. A scalar instruction performs a single operation, while a vector instruction performs a varying number of operations, depending on the value of the vector length (`VL`) register. Column four presents the number of operations performed by the vector instructions (note that the scalar operation counts are identical to the scalar instruction counts). Fifth column is the percentage of vectorization of each program. We define the degree of vectorization of a program as the ratio between the number of vector operations and the total number of operations performed by the program (i.e., column four divided by the sum of columns two and four). Finally column six presents the average vector length used by vector instructions, and is the ratio between vector operations and vector instructions (columns four and three, respectively).

What can be seen for table 1 is that the eleven programs have wildly different characteristics, ranging from programs that are almost 100% vectorizable, like SWM256, to programs that exhibit almost no vectorization at all (ORA). All tables that will be presented in this paper have been sorted following the degree of vectorization column in table 1. We have found this measure very useful in classifying the programs into distinct categories. In fact, the degree of vectorization measured in terms of operations is more accurate than other indirect measures of vectorization such as speedup. We have the execution time of the eleven programs when run in vector mode and scalar mode, and have found that the associated speedup does not have a good correlation with the degree of vectorization. Thus, we will use the degree of vectorization in terms of operations as our main measure of vectorization ratio. Table 1 is divided in two regions. The top eight programs programs that should execute efficiently on vector architectures, given that they have a degree of vectorization over 70%. This group of programs will are referred as the "highly vectorizable" set of programs. The next three programs, DODUC, FPPPP and ORA, are almost non-vectorized at all. They represent a class of programs that are not expected to benefit when executed on a vector architecture. Interestingly enough, we have no "medium vectorized" codes. Similar works in the literature [10, 18] usually report three types of programs (high-, medium- and low- degree of vectorization) but the SPECfp92 suite seems to only have programs from the extremes.

Another interesting point from table 1 is the average vector length observed in the programs (section 5 will present more detailed data about vector lengths). For the highly vectorizable subset of programs, the average vector length is quite high for four programs (SWM256, HYDRO2D, SU2COR and TOMCATV) and for the other four barely reaches half of the vector register size (Note that the C34 has a maximum vector length of 128 elements). It is interesting to remark that the average vector length is not heavily related to the percentage of vectorization. The three benchmarks having a percentage of vectorization around 70% have clearly smaller vector lengths than the other five, but there is no general ordering among the benchmarks. Program NASA7 behaves rather differently with respect to the average vector length due to the fact that it is a collection of seven different kernels.

In table 2 we have included the basic operation distribution for all the benchmarks. We have classified all operations executed by each programs in two major groups (Vector and Scalar) and each group has been further broken down into a total of four categories. These four categories are: Load, Store, Mul and Add operations. In the vector group, the Load and Store categories include, respectively, the gather and scatter instructions. The Mul category includes all types of instructions that can only be executed on the general purpose functional unit, that is, includes all multiplications, divisions and square roots. The Add category includes all other arithmetic operations not included in the Mul category, that is, adds, subtracts, logicals, shifts, etc.

What can be seen from table 2 is that most programs under study are memory bound when run on an architecture like the C34, having only one memory bus and two functional units. Since we only have one memory port, all operations listed under columns Load and Store will use the same hardware resource, while operations listed under columns Mul and Add can potentially run in parallel. For all highly vectorizable codes except MDLJDP2 and MDLJSP2 we find that the most loaded resource will be the single memory port. For the highly vectorizable set of programs, it is interesting to note that the scalar portion of the code is clearly much smaller than the vector portion. The total number of memory accesses performed in scalar mode is not very high, and, usually the majority of the scalar code are instructions in the Add category. This is not surprising, since the scalar code present in vector codes is mainly support code. That is, the scalar code found in vector loops is mostly composed of address computations (additions and shifts) and loop control instructions (decrements and comparisons). Nevertheless, WAVE, MDLJDP2 and MDLJSP2 present a different situation since, although their scalar portion of code does not weight very much in the overall operation count, their scalar memory references take up a high proportion of all the program's memory references. For WAVE, scalar references are a 25.4% of all memory references, for MDLJSP2 this percentage is 51.3 and for MDLJDP2 it's 55.0.

4 Operation Frequency

To better understand the distribution of instructions performed by the eleven programs, we have further classified all vector operations into 11 subsets. Table 3 presents this classification. We have taken all vector operations for each program and classified them between memory and compute operations. The compute operations appear in columns 2 to 8 in table 3, under the label "Compute". The memory operations appear in columns 9 to 12 under label "Memory". The first column of the table is the total number of operations (in millions) performed by vector instructions (and is the same as the fourth column in table 1). All the other columns are percentages taken over this first column, and thus, columns 2 to 12 add up to a 100%.

In the compute category, we have distinguished between seven types of operations. The two most common operations are additions and multiplications (subtractions are included in the "+" category). As it can be seen, the most common situation for all programs is to have a 90% of its compute operations split between additions and multiplications. This is the case for SWM256, NASA7, SU2COR, TOMCATV and WAVE. Overall, in these five programs, additions and multiplications appear to be in 1 to 1 proportion. For the other three programs, we have that a high proportion of all operations is listed under column "other" (23% for HYDRO2D, 37% for MDLJSP2 and 40.6% for MDLJDP2). Investigation of these three programs has led to the following conclusions. In programs MDLJDP2 and MDLJSP2 (quantum chromodynamic benchmarks), the bulk of the calculations is performed in routines JLOOPU and JLOOPB. Briefly, these two routines (re)compute the force interaction between particles and use and build (resp.) a dynamic neighbor list of particles. In the force computation, a particle to particle center separation computation is carried out and this computation requires a great deal of "IF" constructs. These IF's get compiled into a long sequence of mask building instructions and vector comparison instructions. These two instructions (`mask` and `le`) are the ones that dominate column "other" in table 3. Program HYDRO2D (hydro-dynamical Navier-Stokes equation solver) spends most of its time in routine FILTER, where the anti-diffusion fluxes are limited. This routine is composed of a set of loops that compute differences in the Z- and R- directions using "IF's" and "OR's" followed by a set of loops where MAX/MIN computations are carried out to actually limit the fluxes. Both the IF's, OR's and MAX/MIN make heavy use of `mask`, `or` and comparison instructions.

The other types of compute operations are much less frequent. Nevertheless, it is very interesting to see in table 3 the percentages for high-cost operations like division and square root. Most usually, hardware implementations of division and square root are not full pipelined, and thus, may require a high number of cycles between consecutive dispatching of the same operation. For example, consider that a square root requires 20 cycles to complete and that it is not pipelined (that is, a second square root operation will have to wait the full twenty cycles before starting. Then, for example, the importance of the 0.33% of square root operations performed by WAVE is

Program	#bbs	#insns S	#insns V	#ops V	% Vect	avg. VL
SWM256	1.95	6.20	74.51	9534.34	99.9	127
HYDRO2D	4.77	41.50	39.15	3973.78	99.0	101
NASA7	17.12	152.42	67.26	3911.86	96.2	58
SU2COR	22.85	152.55	26.78	3356.83	95.7	125
TOMCATV	19.91	125.75	7.19	916.80	87.9	127
WAVE	50.59	676.52	41.27	1807.23	72.8	43
MDLJSP2	136.28	1427.53	83.50	3832.95	72.9	45
MDLJDP2	125.95	1495.87	89.05	3731.33	71.4	41
DODUC	100.05	1109.19	31.18	123.84	10.0	3
FPPPP	127.40	5070.77	39.31	102.12	2.0	2
ORA	292.75	2820.37	0.00	0.00	0.0	3

Table 1: Basic operation counts for the SPECfp92 programs.

	Types of operations							
	Vector				Scalar			
Program	L	S	*	+	L	S	*	+
SWM256	3153.52	1344.92	2045.53	2990.37	0.14	0.07	0.02	4.04
HYDRO2D	1318.32	452.37	630.51	1572.57	10.98	3.03	0.90	22.15
NASA7	1069.07	652.53	1071.17	1119.09	51.88	21.33	2.88	61.48
SU2COR	987.80	506.97	926.39	935.68	42.14	23.84	0.35	66.08
TOMCATV	337.88	175.65	175.68	227.59	13.36	0.31	0.00	92.25
WAVE	546.65	382.51	386.99	491.09	211.29	105.19	67.46	246.55
MDLJSP2	344.03	470.41	660.58	2357.92	640.51	218.82	0.05	461.47
MDLJDP2	333.43	447.27	599.54	2351.10	690.53	264.55	0.06	444.91
DODUC	39.71	22.65	23.33	38.16	251.24	132.61	110.01	524.16
FPPPP	11.47	19.68	26.41	44.56	1909.11	612.89	876.17	1559.35
ORA	0.00	0.00	0.00	0.00	851.81	344.74	460.10	912.46

Table 2: Balance between Vector and scalar operations.

automatically multiplied by 20, since each individual square root operation costs the same as, say, performing 20 simple additions. Thus, the percentages represented in columns 4 and 5 would become more important.

Under columns labeled $\prod$ (prod) and $\sum$ (sum) we have included the frequency of operations performed using the reduction operators available in the C34 architecture. In terms of operation complexity, the prod and sum operators are very similar to the multiplication and addition operations. The major difference is that these reduction operations have a higher latency since they have to retrofit its own results into the adder/multiplier in order to perform the full reduction. From table 3 it is clear that the prod operation is seldom found in the SPECfp92 programs while the sum operation appears in some programs (MDLJSP2 4.03 % and MDLJDP2 3.67%).

In the memory category, we have included vector loads and stores as well as the indirect memory accessing functions, gather and scatter. With respect to loads and stores, we have that overall the ratio between loads and stores is 1.87, making loads be almost twice as frequent as stores. It has already been reported for scalar programs [6] that it is more common to have load instructions than store instructions. Also in [9] a similar relation was found for code compiled on a Cray Y-MP vector multiprocessor. It is interesting to note that for all programs the percentage of store operations remains between a 10% and a 20%.

Program	Total ($\times 10^6$)	Compute							Memory			
		+/−	×	÷	$\sqrt{}$	$\prod$	$\sum$	other	LD	ST	gath.	scatt.
SWM256	9534.34	31.35	20.63	0.83	0.00	0.00	0.00	0.01	33.08	14.11	0.00	0.00
HYDRO2D	3973.78	15.41	13.66	2.09	0.12	0.00	0.45	23.67	33.18	11.34	0.00	0.04
NASA7	3911.86	27.26	26.67	0.66	0.05	0.00	0.13	1.22	27.33	16.68	0.00	0.00
SU2COR	3356.83	22.99	26.46	0.99	0.15	0.00	0.55	4.33	24.65	15.10	4.78	0.00
TOMCATV	916.80	24.12	18.45	0.71	0.00	0.00	0.00	0.71	36.85	19.16	0.00	0.00
WAVE	1807.23	18.33	20.48	0.61	0.33	0.00	0.43	8.39	26.26	21.17	3.98	0.00
MDLJSP2	3832.95	19.92	16.06	1.18	0.00	0.00	4.03	37.23	6.67	11.87	2.31	0.40
MDLJDP2	3731.33	18.44	14.98	1.09	0.00	0.00	3.67	40.66	6.55	11.70	2.39	0.29
DODUC	123.84	15.19	16.00	2.72	0.12	0.00	0.15	15.35	32.07	18.29	0.00	0.00
FPPPP	102.12	17.26	23.57	1.91	0.00	0.39	2.41	16.75	11.23	19.27	0.00	0.00
ORA	0.00	0.00	20.00	0.00	0.00	0.00	0.00	0.00	20.00	60.00	0.00	0.00

Table 3: Breakdown of Vector operations in the SPECfp92 programs. Column Total is in millions of operations. Rest of columns are percentages over column Total.

5 Vector Length Distributions

A very important factor in achieving high performance in vector execution is the effective vector length used by vector instructions. Current vector architectures have very different sizes for its vector registers, ranging from 64 elements in Cray-YMP, to 128 elements in YMP-C90 and Convex C3 and up to 8192 elements in the reconfigurable vector register file of Fujitsu's VP2400/10 machine. The higher the number of register elements used during the computation the better, because memory and functional unit latencies and vector start-up costs are amortized over the length of the vector register. Longer vector lengths have a lower associated overhead and, thus, achieve higher performance.

We have monitored every single value stored in the vector length register of the Convex C3 machine. Recall that this machine has a maximum vector length of 128. The accumulated percentage distribution of all the values gathered is plotted in figure 1. This curve is labeled DEFINED. We have also computed a weighted version of this distribution that corresponds to weighting the percentage of times that each vector length is defined by the number of vector instructions that will actually use that vector length. This second curve is labeled USED.

From figure 1 what is clear is that the vector length distributions do not follow any regular pattern. Three programs behave as expected (SWM256, SU2COR and TOMCATV), having the majority of their vector lengths clustered around 128 and having a small percentage (below 5%) of very short vector lengths that are the residuals generated due to strip-mining. Hydro2d has a single dominant vector length (102) that comes from the fact that this value is the number of grid points used in the z-direction of the problem. Two programs, NASA7 and WAVE, have a distribution that follows a staircase, having several dominant vector lengths. The MDLJSP22 and MDLJDP22 programs have obviously very similar distributions, with almost 90% of all their vector lengths being less than 64 elements. Finally DODUC, FPPPP and ORA have the shape of non-vectorizable programs, with almost no vector lengths larger than 10 elements.

All in all, this data suggests that even among vectorizable programs the utilization of the vector registers of the C34 machine varies a lot. While some of the programs almost always use the 128 elements, the others could also run at a similar performance on machines having shorter vector registers. Next section will discuss in more detail what could be a better choice for the length of the vector registers.

5.1 Usage of the Vector Registers capacity

Given the results presented in the previous section, we want to investigate what would be a reasonable value for the length of the vector registers. What we have done is compute the

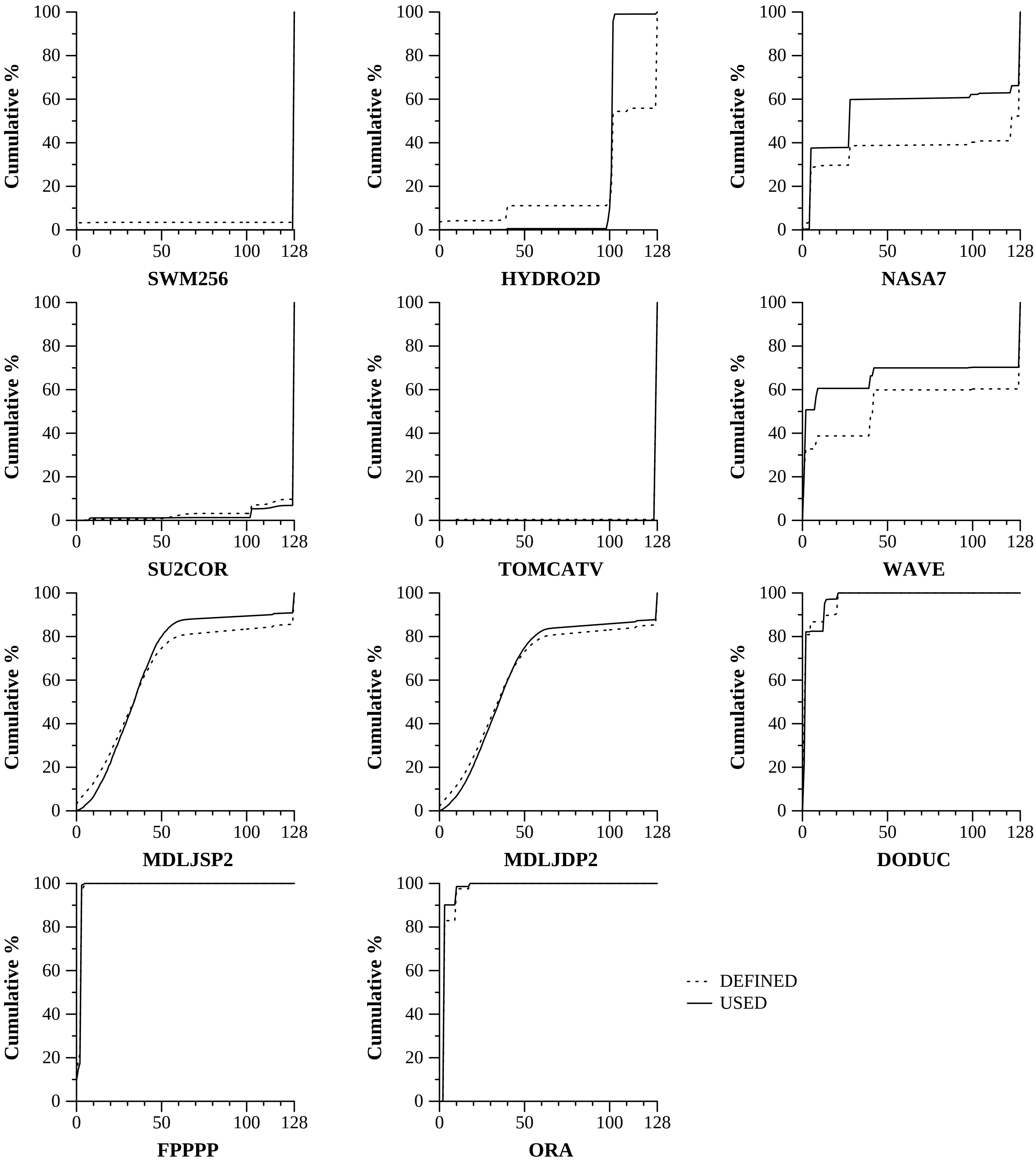

Figure 1: Distribution of the VL values for the SPECfp92 programs

average data fill of the vector registers. For example, say a program has two vector instructions, and that the first instruction uses a vector length of 128 and the second one a vector length of 31. If we consider the vector register size to be 128, we have that the first vector instruction is filling with useful data a 100% of all the vector register slots, while the second instruction is only filling the $31/128 = 24.21\%$ of all the vector register slots. The average filling would be 62.11%. Now, if we consider a vector register size of 64 elements, it is clear that the same two instructions would have to execute twice in order to get all their work done. The first instruction would "translate" into two instructions that would operate on 64 elements each one. The second instruction would not change and would operate on 31 elements. Thus, if we had a vector register of 64 elements, 2 instructions would fill the vector registers at a 100% and one instruction would be filling only a 48.44%, yielding an average fill of 82.81%.

Figure 2 presents the percentage of filling of each program, considering four possible values for the vector register size: 16, 32, 64 and 128 elements. For example, the figure shows how in NASA7, with a register size of 128, the average filling of a vector register is around 45% of its maximum capacity (i.e., around 57 elements). For the same program, if the vector register size was 16 elements, almost all vector instructions would fill around 90% of each vector register used.

From figure 2 we can see that the degree of filling is satisfactory for four programs (SWM256, HYDRO2D, SU2COR and TOMCATV), and that NASA7, WAVE, MDLJSP2 and MDLJDP2 are wasting more than 50% of the capacity of the 128-element register. For these four programs, reducing the register size to 64 elements would result in an increase of utilization of the register capacity of more than 20 percentage points. The data presented in figure 2 suggests that an interesting tradeoff to be studied should be the effect of reducing the number of elements in the vector register size. This reduction could be dangerous in terms of performance since it would imply (a) executing more instructions and (b) paying more times the memory latency, but could be compensated in terms of total economical cost or by using the extra available area for introducing more vector registers, thus reducing the amount of spill code.

6 Spill Code

Another interesting aspect of vector processors is the usage of the vector register file. Since each vector register in a vector processor has a long size, the total amount of vector registers that can be implemented in a vector architecture is not very high. The C34 and the CRAY Y-MP, for example, only have 8 vector registers. This limited number of registers makes very difficult the task of the compiler, since it has to balance aggressive scheduling techniques against a very limited number of vector registers. When compiling large dependence graphs, the register allocator usually runs out of available free registers and has to spill some live variables to the stack and later reload them. This spill code will be very negative in an architecture where there is only one memory bus. Every spill load/store instruction inserted by the compiler will result in a net increase of total memory traffic. Since we already saw in section 3 that the benchmark programs are memory bound, this net increase in memory traffic will most probably not be hidden by other computations and will result in a decrease in performance.

The compiler used to compile the SPECfp92 codes puts all its spill code on the stack, and seldom ever uses the stack to hold vector data that it is not spill data. Thus, we have been able to distinguish spill memory operations from "normal" memory operations by checking if the memory access was to the stack or not. We have further enhanced our technique to detect spill code by creating a table (at runtime) of all data that is stored to the stack and checking that there is some load that later reloads it. If this is the case, then we annotate that both the store and the load were spill operations. If, by chance, there is a load or a store that references the stack but it is not in our table, this memory access is not considered to be spill code.

Using this technique, the total number of vector spills has been computed for each program. Table 4 presents the results (all data is in millions of operations). Under column LD we have the total number of non-spill load operations performed in vector mode. Under "spill-LD" we have the total number of spill

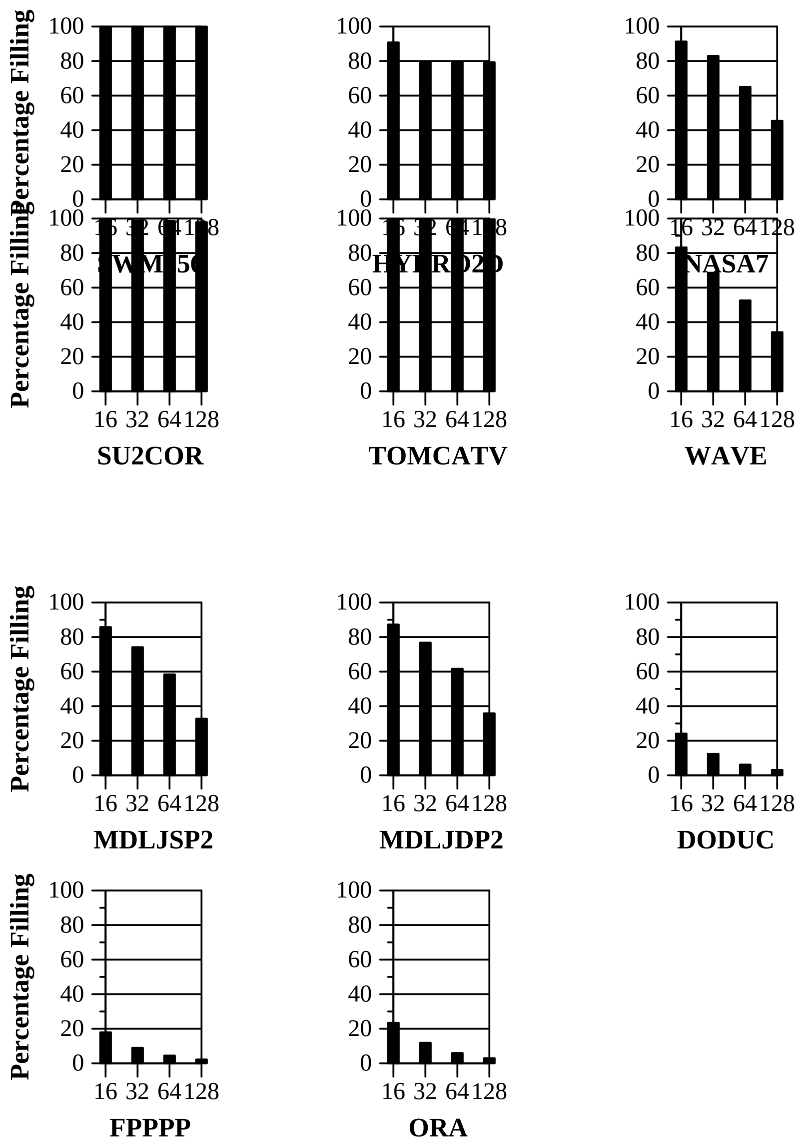

Figure 2: Percentage of filling of the vector registers for different vector register sizes

load operations performed in vector mode. For example, TOMCATV has a high proportion of spill loads over normal loads, since it performs 104 millions of spill-loads and 233 millions of normal loads, which represents a $(104 \div (104+233))$ 31% approximately of spill-loads.

It is interesting to note that for the vectorizable programs the percentage of spill is rather remarkable. Considering loads and stores together, TOMCATV has a 40.5% of spill code, MDLJDP2 a 30.3%, MDLJDP2 a 27.3% and SU2COR a 20.3%. Inspection of the TOMCATV code reveals that this program spends most of its time in a doubly nested loop where the residuals to be used in the subsequent tridiagonal solving are computed. This residual computation uses up to 12 different array variables, that, as expected, do not fit in the eight vector registers available. Thus the compiler inserts all the necessary spill code (16 spill-stores and 16 spill-loads) to fit the 12 variables into the available registers. All in all, the influence of having only eight registers in a single bus architecture is quite evident. All the spill code presented in table 4 will impact (negatively) the run time of the programs

7 Pressure on the dispatch unit

A very interesting property of vector architectures is that they can potentially achieve the same amount of instruction level parallelism as superscalar architectures using a much simpler decoding hardware. Superscalar architectures have to explore a "window" of ready instructions to try to find independent instructions and dispatch them. On the other hand, vector processors simply issue instructions in a sequential way but, since each vector instructions implies a lot of operations, each vector instruction runs for a long period of time. When the vector processor issues two vector instructions, it has two functional units busy at the same time, much as a superscalar processor does, but with a much smaller effort in terms of instruction bandwidth and dispatch logic.

To get an insight into the amount of work that the dispatch unit of a vector processor has to do per cycle, we have taken the following approach. First, we compute a lower and an upper bound of the execution time of each program (in this section we understand "execution time" as "number of cycles executed"). As a lower bound of the number of cycles that a program will execute, we use tables 1 and 2 to compute which resource in the architecture is the resource most used. For example, for program SWM256, the memory bus is the most used resource, and, in particular, it is used to perform 4498.44 millions of transactions. This number is the sum of all vector loads and vector stores, and does not include the scalar memory operations. We do not include the scalar memory operations because we are computing an optimistic lower bound and thus we assume that all scalar memory operations hit in the cache. As another example, for program ORA, the most used resource is the dispatch unit itself, since the program has a total of 2820.37 millions of instructions to dispatch and all the functional units of the architecture execute less operations (the adder in the scalar unit performs 851.81 millions of operations, for example).

As an upper bound to the execution time of the program, we take the sum of all its operations. We consider this value to be an upper bound because it is unlikely that the program has no instruction level parallelism at all, and, thus, it is likely that it can overlap some computations with others.

With this two bounds, presented in columns 1 and 2 in table 5, and knowing the total number of instructions issued by the program (column 3 of table 5), we can compute the ratio of instructions per cycle that have to be dispatched by the control unit. By dividing the lower bound of execution time by the number of instructions in the program we obtain an upper bound on the issue ratio. Similarly, by dividing the upper bound of execution time by the number of instructions in the program we obtain a lower bound on the issue ratio. This bounds can be seen in columns four and five of table 5.

Table 5 shows how the subset of highly vectorizable programs does not have very tight bounds. For example, in the worst case, SWM256 has to dispatch 1 instruction every 55.7 cycles. Clearly this does not represent a very high pressure on the dispatch unit. Nevertheless, as the vectorization ratio decreases, the programs start to increase their instruction

Program	Vector load ops			Vector store ops		
	LD	spill-LD	%	ST	spill-ST	%
SWM256	2839.01	314.51	9.97	1030.42	314.51	23.38
HYDRO2D	1297.11	21.22	1.61	431.15	21.22	4.69
NASA7	1048.14	20.93	1.96	632.12	20.41	3.13
SU2COR	786.46	201.34	20.38	404.25	102.72	20.26
TOMCATV	233.84	104.04	30.79	71.61	104.04	59.23
WAVE	401.14	145.50	26.62	318.00	64.50	16.86
MDLJSP2	232.86	111.18	32.32	359.24	111.18	23.63
MDLJDP2	214.88	118.55	35.56	328.71	118.55	26.51
DODUC	22.38	17.33	43.63	11.97	10.68	47.16
FPPPP	5.32	6.15	53.59	13.53	6.15	31.23
ORA	0.00	0.00	0.00	0.00	0.00	0.00

Table 4: percentage of spill code in vector memory operations. Columns 2, 3, 5 and 6 are in millions of operations.

Program	cycles (lower bound)	total ops.	total insns	(1)/(3)	(2)/(3)
SWM256	4498.44	9540.54	80.71	55.7	118.2
HYDRO2D	1770.69	4015.28	80.65	22.0	49.8
NASA7	1721.59	4064.28	219.68	7.8	18.5
SU2COR	1494.77	3509.39	179.33	8.3	19.6
TOMCATV	513.53	1042.54	132.94	3.9	7.8
WAVE	929.16	2483.76	717.79	1.3	3.5
MDLJSP2	2357.92	5260.48	1511.04	1.6	3.5
MDLJDP2	2351.10	5227.20	1584.91	1.5	3.3
DODUC	1140.37	1233.03	1140.37	1.0	1.1
FPPPP	5110.08	5172.89	5110.08	1.0	1.0
ORA	2820.37	2820.37	2820.37	1.0	1.0

Table 5: Pressure on the dispatch unit. First column is an ideal lower bound on the number of cycles needed to execute each program. This lower bound is computed for an hypothetical vector architecture that had one Load/Store bus, one adder, one multiplier and a pipelined scalar unit. The bound can be derived from the last eight columns of table 1.

fetch/dispatch bandwidth. The lower six programs in the table will clearly need a fast control unit to avoid wasting many cycles. These six programs will be negatively affected by delay cycles introduced by jumps and i-cache misses, for example. The table also shows the great advantage of vector execution against superscalar execution. For vector programs, the control unit has many cycles where it is idle, and can use them to implement techniques that improve overall performance (branch prediction, out-of-order execution, etc.) at a low cost. These techniques are usually costly in superscalar processors because they have to be implemented (usually) in a single cycle and, thus, can negatively affect the cycle time of the processor. On the other hand, on a vector architecture the same techniques could be implemented without necessarily having to being done in a single cycle.

8 Conclusions and Future Work

In this paper we have used a trace driven approach to gather data about the instruction level characteristics of vector programs. Our method has given us the full execution of the programs, including not only user code (like in many of the instruction level studies found in the literature) but also the library code with all the specific fortran math vector routines.

Using these traces we have presented the degree of vectorization in terms of operations of all the SPECfp92 programs. We have seen how the proportion between compute and memory operations is close to 1 to 1. This implies that having only one memory bus will probably not be enough to feed a pair of functional units. We have also seen how the effective vector length used in the operations varies a lot between benchmarks, with a few of the vectorizable programs having a mean close the the maximum (128), and the rest of vectorizable codes being around 40-50. For this latter group of programs, the startup overhead of vector operations is not totally being compensated along the full length of the vector and they would probably suffer under long memory latency conditions. From the operation type distribution we have concluded that the two most important computation instructions are additions and multiplications (80% of all computation operations) and that they appear with almost the same frequency, except in two concrete benchmarks where heavy use of masking is made. Overall, data suggests that designs that have a good balance between the number of units that perform additions and the number of units that perform multiplications will better match the behavior of the programs.

We have examined the amount of spill code and have found that for the vectorizable set of programs it was quite high. Since the architecture under study is single-ported to memory, we conclude that this spill will have a clear negative effect on execution time. We have also looked at the pressure on the dispatch unit. We have found that for the non-vectorizable set of programs, a dispatching of one instruction per cycle is needed. On the other hand, the vector programs can spend a mean of 12 cycles between instructions without requiring the intervention of the dispatch unit.

The data presented in this paper will be used to study new alternative vector architectures that better match the characteristics of vector programs. We are currently investigating using decoupling techniques to reduce the imbalance between memory and compute operations detected.

References

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

[2] J. J. Dongarra. Performance of various computers using standard linear equations software in a fortran environment. Technical Report CS-89-85, Univeristy of Tennesse, 1993.

[3] 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.

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

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

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

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

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

[9] 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.

[10] 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.

[11] Cray Research, Inc. *Cray Y-MP System Architecture for Systems Analysts*, trysasa revision e edition, April 1990.

[12] Willi Schonauer and Hartmut Hafner. Supercomputers: Where are the lost cycles ? *Supercomputing*, 1991.

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

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

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

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

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

[18] 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.