An Empirical Study of the CRAY Y-MP Processor using the PERFECT Club Benchmarks

Sriram Vajapeyam
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Wei-Chung Hsu
Cray Research, Inc.
Development Division
Chippewa Falls, WI 54729

ABSTRACT

Characterization of machines, by studying program usage of their architectural and organizational features, is an essential part of the design process. In this paper we report an empirical study of a single processor of the CRAY Y-MP, using as benchmarks long-running scientific applications from the PERFECT Club benchmark set. Since the compiler plays a major role in determining machine utilization and program execution speed, we compile our benchmarks using the state-of-the-art Cray Research production FORTRAN compiler. We investigate instruction set usage, operation execution counts, sizes of basic blocks in the programs, and instruction issue rate. We observe, among other things, that the vectorized fraction of the dynamic program operation count ranges from 4% to 96% for our benchmarks. Instructions that move values between the scalar registers and corresponding backup registers form a significant fraction of the dynamic instruction count. Basic blocks which are more than a hundred instructions in size are significant in number; both small and large basic blocks are important from the point of view of program performance. The instruction issue rate is less than 0.5 instructions per clock cycle for each of our benchmark programs, suggesting that the instruction issue stage of the CRAY Y-MP is, on the average, not a bottleneck.

1. INTRODUCTION

An understanding of instruction-level program behavior is essential in the architecture design process. However, experimental studies which collect data that provide such an understanding have, to date, been published for only a few machines. Studies of a non-vector CISC architecture, the VAX [4, 6], a couple of microprocessors, the Intel 8086 [3] and the MC68020 [15], and a current RISC architecture, MIPS [7], have been published, among a few others. Detailed studies of vector processors, using as benchmarks long-running programs compiled aggressively for performance, have not been reported.

In this paper we present a study of the CRAY Y-MP [2], using as benchmarks the PERFECT Club [5] set of application programs. The programs are compiled by the Cray Research production FORTRAN compiler, CFT77, version 3.0, for a single processor of the CRAY Y-MP. We report and analyze dynamic instruction and operation frequencies, frequencies of basic blocks of various sizes, and instruction issue rates. The data shed light on the use of a vector processor by a highly optimizing compiler.

In section 2 we describe our measurement methodology, our benchmarks, and some caveats that should be observed in using the data reported. In section 3 we discuss the usage of various instructions by the programs. In section 4 we discuss the sizes and the vectorization of basic blocks of our benchmark programs. In section 5 we briefly discuss instruction issue rates. In section 6 we draw some conclusions from the study.

2. STUDY ENVIRONMENT

2.1. Measurement Methods

We use two methods of data collection in our study. First, most of the data presented in this paper are dynamic counts such as instruction frequencies, basic block sizes, etc., which do not involve measurement of the *time* taken by the processor to execute the program. Such data can be collected very quickly by the following simple, widely-used technique. Programs are composed of basic blocks of instructions, each basic block being a maximal sequence of instructions with a single entry point (the first instruction) and a single exit point (the last instruction). The branch instructions in the program determine the number of times each basic block in the program text is executed, and the sequence in which the basic blocks are executed. Statistics that are affected only by the frequency of execution of individual basic blocks, and not by the sequence in which they are executed, can be gathered easily and quickly by first collecting data for the individual basic blocks in the program and then scaling the statistics for each basic block by the execution frequency of that basic block. The dynamic execution frequencies of the basic blocks need to be collected first; instrumentation of programs to collect this information is routinely done (for example, [14]). We use a Cray Research production software tool, JUMPTRACE [11], to obtain execution frequencies of the basic blocks in our benchmarks. Another software tool analyzes the basic blocks in CRAY machine code and uses the basic block execution frequencies to scale the data collected for each basic block.

Second, we present some data regarding the time taken for program execution. Such data are collected using the Hardware Performance Monitor (HPM) available on the CRAY Y-MP. The HPM is a set of hardware counters that can be turned on during program execution to collect certain statistics in an unobtrusive manner. For example, HPM monitors program execution time, instruction issue stage utilization, and the number of floating-point, integer, and memory operations executed.

© 1991 ACM 0-89791-394-9/91/0005/0170 $1.50

2.2. Benchmarks

We use the PERFECT Club [5] programs as benchmarks in this paper. Briefly, the PERFECT Club benchmark set is the result of a large-scale benchmarking effort toward aiding supercomputer evaluation, and comprises of thirteen long-running supercomputer *application* programs chosen to represent the spectrum of characteristics of scientific applications. These benchmarks are becoming widely accepted as standard benchmarks for supercomputer evaluation.

Program performance on any system is determined by the abilities of both the compiler and the processor. We compile our programs, for a single processor of the CRAY Y-MP, using the Cray Research production FORTRAN compiler, CFT77, version 3.0. CFT77 is an aggressive compiler that optimizes code and vectorizes it for high performance. For example, the compiler makes special efforts to hide scalar memory latency: memory load instructions for operands that are consumed in future loop iterations are issued towards the end of the current loop iteration to hide memory latency. Since we study compiled code, we view our study as one of the PERFECT Club benchmarks executing on the CRAY Y-MP *system* comprising a state-of-the-art vectorizing and optimizing compiler, and the fine-grain parallel CRAY Y-MP processor.

In this paper we study the user routines of the benchmark programs. We do not include in our study the library routines executed by the benchmark programs. A practical reason for this is the fact that the basic block profiler currently available to us does not profile library routines[1]. However, a more important reason is the fact that many of the CRAY library routines are hand-coded in assembly language *due to performance considerations*. This implies that library routines will utilize the processor better than compiled code. By not including the library routines in the study, we focus here on the performance of compiled FORTRAN code.

Table 1 presents the execution-time, in CRAY Y-MP clock cycles or pulses (CPs), and the number of instructions executed for each of the benchmark programs. These numbers are for the execution of the entire program, *i.e.*, they include both user and library routines. We observe that each program executes hundreds of millions of CRAY Y-MP instructions, and takes hundreds of millions of clock cycles to run. Thus, the benchmarks we study are long-running applications, as opposed to the kernels or small benchmarks that are commonly used for vector machine studies.

2.3. Caveats

All the data collected for the CRAY Y-MP using our technique of profiling at the basic block level are accurate, but for one exception that is due to the vector architecture of the CRAY Y-MP. A vector instruction executes identical operations on a number of data elements, the number being determined at run time, by the contents of a special Vector Length (VL) register. Since we do not simulate program execution, the content of the VL register during the execution of each individual vector instruction is not available to us. Assuming that each vector instruction operates on the maximum vector length of 64 elements results in over-

[1]The current version of the profiler instruments code during compilation; the library routines, being either hand-coded in assembly language or being compiled directly into object form, are therefore not instrumented.

Benchmark	Time (millions of CPs)	Instructions (millions)
ADM	4,492	1,597
ARC3D	5,743	1,218
BDNA	1,550	300
DYFESM	647	217
FLO52	801	176
MDG	9,048	1,780
MG3D	8,750	911
OCEAN	8,008	3,387
QCD	2,297	896
SPEC77	3,131	550
SPICE	1,585	418
TRACK	1,719	500
TRFD	1,304	503

Table 1. Benchmark Sizes
(as reported by the Hardware Performance Monitor)

estimation of several measurements; in reality, several vector instructions operate on many fewer data elements. To overcome this problem, we use the average vector lengths reported for three different vector instruction classes — floating-point, integer, and memory instructions — by the HPM, for each program. The data collected using the average vector lengths are far more accurate than data obtained using the maximum vector length. However, there still exists a small margin for error, due to problems associated with using averages of numbers. One, the average vector lengths of the individual instructions that form each instruction class could be significantly different from the average for the whole class. Two, the average vector lengths reported by HPM are averages over the execution of the entire program, while we use these averages to study only the user routines of the program. The average vector length of the user routines could be quite different from the average for the library routines, and thus quite different from the average for the entire program execution.

Investigation of related program measurements[2] (some of which are not reported here) and investigation of important basic blocks of some of the benchmark programs lead us to believe that the error introduced by this approximation is not significant, at least for the study reported in this paper. However, one should keep this assumption in mind when using certain results.

We note in passing that simulating the programs in their entirety would require enormous amounts of CPU time, since simulations can be two to three orders of magnitude slower than the programs being simulated, and the

[2]For example, we use a methodology (not reported here) for quickly estimating the *execution times* of programs. The methodology is accurate to within 10% for most programs when the average vector lengths are used; the accuracy is much lower when each vector instruction is assumed to operate on 64 data elements. A large part of the 10% error is due to factors such as simplistic modeling of the complex CRAY Y-MP memory system; the error introduced by the problems associated with using only averages of vector lengths is insignificant.

programs themselves have long execution times (Table 1). The assumptions made about vector lengths in order to avoid a full simulation of the benchmarks are thus justified. The assumptions help us collect data relatively quickly, and at the same time do not cause significant errors in the data collected.

3. INSTRUCTION AND OPERATION MIX STUDIES

We present in this section several measurements pertaining to the CRAY Y-MP processor instruction set usage, program vectorization, and operation execution counts.

3.1. Overview of the CRAY Y-MP Processor

We first briefly highlight some features of the CRAY Y-MP processor architecture [2], to provide background for the discussion of our measurements. The processor architecture of the CRAY Y-MP is very similar to that of the CRAY X-MP [1]; a major difference is that the CRAY Y-MP processor can address a larger memory space (32 address bits in the Y-MP versus 24 bits in the X-MP).

The processor can be partitioned into vector and scalar portions; the memory interface of the processor consists of four ports: three for data transfers, and one for I/O and for fetching instructions. The vector and scalar portions of the processor share floating-point functional units, but have separate functional units otherwise. The scalar portion can be viewed as consisting of an address-computation unit (an address unit, henceforth) and a scalar-computation unit, each having its own set of functional units. The processor has three register sets — a set of eight vector (V) registers with 64 elements each, a set of eight scalar (S) registers, and a set of eight address (A) registers. Furthermore, the S and A register sets have corresponding backup register sets, T and B, of 64 registers each. A backup register is used to temporarily hold values when the corresponding primary register set is full and a register needs to be spilled to make room for another value. The functional units of the processor are fully pipelined.

The CRAY Y-MP processor has one-parcel (16 bits), two-parcel, and three-parcel instructions. The architecture is a LOAD/STORE architecture — memory is accessed explicitly, and only by data-transfer instructions. All computation instructions are register-register instructions. The processor can issue one-parcel instructions at a peak rate of one per clock cycle; two-parcel and three-parcel instructions need two clocks for issue. The processor has an instruction cache (called I-buffers), but no data cache.

Overall, the processor is highly pipelined to exploit fine-grain parallelism. The compiler attempts to identify and increase fine-grain parallelism in the code to take advantage of this hardware. For example, the compiler unrolls loops to increase parallelism, and software pipelines memory operations to tolerate long memory latencies (*i.e.*, it pre-loads in the current iteration some of the memory values used in the next iteration). Increasing parallelism in code, however, results in a need for more registers. For scalar code, the compiler can take advantage of the backup registers provided, to tackle the need for registers. Loop unrolling also results in a need for large instruction caches. The Cray Research compiler limits the amount of loop unrolling to the size of the instruction cache (512 parcels or 1K bytes), to avoid repeated I-buffer misses for loop instructions [16]. (The amount of unrolling is dependent on the size of the original loop body.) All these factors affect the dynamic execution characteristics of an application program; we identify some of the effects of the above factors in the data presented in the next few sections.

3.2. Program Vectorization

The vectorization of a program is important to its performance. Vector instructions execute on a series of data elements in a pipelined fashion; they hide functional-unit latency with pipeline parallelism, and perform better than a corresponding scalar loop (see for example [8], Chapter 7). Thus, the fraction of program execution time spent executing vector instructions is a measure of the efficiency of program execution. This fraction is related to the dynamic frequency of vector *operations*[3] in the program, although the relationship may not be linear, as per our experience. The former is the best metric of the vectorization of a program since it directly represents program performance. However, since we do not study statistics related to execution *time* in this paper, we will rely on the dynamic frequency of operations executed with vector instructions to measure program vectorization.

Folklore has it that many scientific programs have as many as 90% of their operations executed by vector instructions (for example, this is the number quoted along with the Lawrence Livermore Loops [13]). However, scientific programs could be inherently unvectorizable due to recurrences (or, data-dependences across the iterations of a loop), ambiguous array subscripts, data-dependent branches and/or subroutine calls inside loop bodies, etc. Furthermore, current limitations of state-of-the-art compilers prevent vectorization of some code that could actually be run in vector mode if compiled by hand. Here we characterize the vectorization of a scientific workload by a state-of-the-art compiler, to determine the current relative importance of the scalar and vector portions of a supercomputer.

We use the fraction of all program operations vectorized (*i.e.*, executed by vector instructions) as our vectorization metric, since program performance is dependent on the vectorization of all operations. Instead, the fraction of floating-point operations vectorized is sometimes used as the vectorization metric, as done for example in [8]. Let us first examine the vectorization of various classes of operations, for our benchmark programs (Table 2). Note that the fractions presented in Table 2 are fractions of operations and not of instructions. For example, for ARC3D, 99.8% of all floating-point operations are executed in vector mode; these vectorized floating-point operations constitute 45.2% (column 3 of the table) of all operations of the program. The data show a fairly good correlation between the fraction of floating-point operations vectorized and the fraction of all program operations vectorized. (The exceptions to this correlation are TRFD and DYFESM.) For example, for BDNA, 99.4% of all floating-point operations and 96.1% of all operations are vectorized; for TRACK, 11.6% of all floating-point operations and 14.4% of all operations are vectorized. Thus, the vectorization of floating-point operations is a fair indicator of overall program vectorization, for our benchmarks. On the other hand, the fraction of either integer or memory operations vectorized is not very correlated with the fraction of all operations vectorized, and neither is a very good metric of program vectorization. For example, 48.6% of TRACK's memory references are vectorized, but only 14.4% of all of its operations are vectorized. We also observe that the fraction of memory operations vectorized is quite high for ten of the thirteen programs;

[3]We distinguish between operations and vector instructions throughout this paper. A vector instruction executes several operations, one on each vector element. An operation is equivalent to a scalar instruction.

Benchmark	Operation Class						
	Floating-Point		Integer		Memory		All
	% of F.P. Ops.	% of All Ops.	% of Integer Ops.	% of All Ops.	% of Memory Ops.	% of All Ops.	%
BDNA	99.4	52.1	97.1	4.4	98.4	39.6	96.1
MG3D	100.0	33.1	47.9	0.5	99.9	61.5	95.1
FLO52	99.9	42.4	60.9	1.5	99.8	47.7	91.5
ARC3D	99.8	45.2	25.6	0.3	99.7	45.5	91.1
SPEC77	98.3	37.3	74.8	1.9	99.0	51.1	90.3
MDG	95.5	25.6	95.5	20.1	96.0	41.9	87.7
TRFD	99.7	27.5	7.2	0.3	97.8	42.1	69.8
DYFESM	97.3	34.4	36.9	2.5	91.0	31.9	68.8
ADM	69.6	15.1	32.3	3.8	80.8	24.0	42.9
OCEAN	54.8	14.7	0.0	0.0	79.3	28.1	42.8
TRACK	11.6	1.4	1.2	0.1	48.6	12.9	14.4
SPICE	9.6	1.1	0.0	0.0	32.6	10.5	11.5
QCD	6.6	1.3	6.5	0.8	16.5	2.1	4.2
Arithmetic Mean	72.5		37.4		80.0		62.0

Table 2. Percentage Vectorization of Various Operation Classes

vectorized integer operations constitute only a small portion of all operations.

We note that the fraction of all operations vectorized spans almost the entire range for the benchmarks. Insofar as the benchmarks are representative of scientific workloads, we can conclude that the *average* fraction of operations vectorized is much less than the usually assumed 90%. For our benchmark set, the average[4] vectorization is 62%.

Considering the clustering of numbers in the overall vectorization column of Table 2, we partition the programs into three subclasses: *highly-vector*, *moderately-vector*, and *scalar* programs. We will present data only for these three subclasses of programs in the rest of this paper; space constraints and the volume of data prevent us from presenting information for the individual programs. We classify the programs as follows. QCD, SPICE, and TRACK make up the *scalar* benchmarks; ARC3D, BDNA, FLO52, MDG, MG3D, and SPEC77 are the *vector* benchmarks; and ADM, DYFESM, OCEAN, and TRFD make up the *moderately-vector* benchmarks. (DYFESM and TRFD are on the border line between two subclasses; we use information about instruction-issue stall times, presented in section 5, to push them into the moderately-vector category.) We believe that classifying the programs as above minimizes the loss of information caused by considering only averages of numbers, since the groups contain programs with very similar characteristics. Additionally, the classification helps us identify certain characteristics of the three subclasses.

3.3. Instruction Usage and Operation Counts

In this section, we discuss the CRAY Y-MP *instruction set* usage and the *operation* mix in our benchmarks. We determine a benchmark subclass's usage of an instruction by averaging the *normalized* usage of the instruction by each of the programs in the subclass; thus, all the programs of a subclass are given equal importance, irrespective of the number of instructions executed by them individually. Since vector instructions execute several operations each, the count of various operations executed presents a better picture of a program's utilization of machine resources. The numbers for operation usage are computed by expanding each vector instruction into the average number of operations that it executes (as reported by HPM) for each program[5]. Table 3 classifies the operations executed by the benchmarks into various broad operation classes that are present in several architectures. For example, 27.82% and 1.52% of all operations in the moderately-vector benchmarks are floating-point and branch operations, respectively. Data thus classified could be compared to data presented for other machines. Table 4 further subdivides the information presented in Table 3 into classes that correspond to the functional units present in the CRAY Y-MP.

[4]We use the arithmetic mean of the vectorization ratios of the individual benchmarks. This ensures that all programs are given equal weight, irrespective of the number of instructions or operations executed by them individually.

[5]If a vector instruction were replaced by scalar code, the number of operations needed to implement an equivalent loop could be 2×VL or 3×VL, instead of the VL operations needed for the vector instruction. We assume each vector instruction executes VL operations.

Operation Class	Benchmark Subclass		
	Scalar	Moderate	Vector
All FP	14.29	27.82	39.66
All Memory	23.83	35.82	48.40
All Integer	10.73	8.83	5.48
Address Comp.	7.62	6.99	1.82
Miscellaneous	37.57	19.03	4.25
Branches	5.94	1.52	0.34

Table 3. Percentage of Operations in each Operation Class

From Table 3, we observe that the vector benchmarks are almost entirely floating-point operations and memory references. On the other hand, the scalar benchmarks have comparable amounts of floating-point and integer operations; for the moderately-vector programs, floating-point operations are three times as frequent as integer operations. By integer operations we mean only those operations that are executed by the scalar-computation unit. Address-computation instructions, while also being integer operations, are executed by the address-computation unit and are classified separately. However, scalar integer operations are also used to perform some address computation work, since the address unit is only 32 bits wide while the integer data-type supported by the architecture is 64 bits long. For example, when array indices are passed as arguments to subroutines they are stored as (64-bit) integers, and they hence have to be manipulated by the scalar unit. Since there is no easy way to determine whether an operation carried out in the scalar unit is for address computation, we classify all integer operations carried out in the scalar unit as (non-address computation) integer operations.

The address computation instructions, executed in the address unit, are used for generating the memory addresses needed by all scalar memory operations; in addition, they are also used for maintaining loop counters on the CRAY Y-MP. Address operations are comparatively less frequent than scalar integer operations in the scalar benchmarks, while the two are comparable in number for the moderate benchmarks. On the whole, scalar programs can be expected to have a higher proportion of address arithmetic operations since they also have a higher proportion of scalar memory operations. When memory references are vectorized, the vector instruction implicitly does address arithmetic, and hence we see fewer explicit address arithmetic operations for highly vectorized code. The data presented bear this out: scalar programs have about 7.5% address operations, while the vector programs have less than 2%.

Register Transfers

Most strikingly, more than one-third of all *operations* of the scalar benchmarks are operations used to transfer values between the various register sets in the processor (miscellaneous category in Tables 3 and 4). The proportion of these register transfer operations decreases as we move to more vectorized programs; the pressure on the non-vector registers is of course higher in scalar code. The backup registers are used to temporarily hold values whenever the regular register sets are full, resulting in several *register spill* instructions[6]. Another reason for the register-register move instructions is the fact that conditional branches in the CRAY Y-MP are based on the contents of registers A0 or S0. The compiler needs to shuffle registers around so that the condition computed is stored in A0 or S0 before the branch is issued. We observe that about 27% of all operations in the scalar benchmarks are spill operations (20% are spills of the S registers [S_SPILL] and 7% are spills of the A registers [A_SPILL]). The pressure on the S registers is higher than that on the A registers, as is to be expected, since there are more result-computation instructions than address-computation instructions in the programs. Another reason for the lower pressure on A registers is that address computation instructions operate on array addresses and loop counters and these are available in the registers most of the time. Scalar computations, on the other hand, more often use data values loaded from memory, and the latency of memory loads tie up the scalar registers for longer periods of time. Overall, the high amount of spill instructions is due to the fact that eight primary registers are not sufficient to hold frequently-used local data and to support deeply-pipelined functional units at the same time. Furthermore, the compiler unrolls loops to exploit parallelism, resulting in more registers being live simultaneously, which makes the problem worse. We also note that MOV instructions, used to move values between A and S registers, are significant in number. MOV instructions constitute 11% of all operations for the scalar benchmarks, and 6% of all operations for the moderately-vector benchmarks. One could think of the MOV instructions as being akin to moving values between the address and computation units of a decoupled architecture.

Although scalar programs have a high proportion of these miscellaneous operations, the contribution to execution time of these operations could be quite small, and disproportional to their number. The deep pipelines in the CRAY Y-MP for computation instructions cause long waits in the instruction-issue stage for dependences to be resolved. Scalar code has a lot of dependences and hence has frequent instruction-issue stalls. The compiler can hide the cost of the single-cycle register-register move instructions by scheduling them for execution during these data-dependence stalls, thus essentially executing them for free. This is one example of the possibility of a large difference between the dynamic frequency of an instruction and its contribution to program execution time. This difference is all the more important in a machine that has several parallel, pipelined functional units, since several instructions can be executing simultaneously, thus making the attribution of program time to instructions more difficult.

Larger register sets would naturally decrease the number of spill instructions. However, larger register sets would not be worthwhile if they increase the machine clock cycle, especially if the spill instructions incur little cost in execution time anyway. MOV instructions, on the other hand, are mainly a result of the register-file and functional-unit architecture of the processor. The functional units and the register files have been separated into A and S sets to provide more parallel, decoupled execution, and MOV instructions are a necessary part of this separation. Also, in the current Y-MP architecture, the A unit does not have shift, logic, and 64-bit integer calculation functionalities. Therefore, many of the MOV instructions move data from

[6]Register spilling refers to moving a temporary value in a register to memory, to make room in the register set for another value that will be accessed before the value being spilled. In the CRAY Y-MP, register spilling usually only results in a movement of values between the primary (A & S) registers and the backup (B & T) registers, and we term these moves *spill instructions*.

Operation Class	Instruction Class	Benchmark Subclass					
		Scalar		Moderate		Vector	
		Insts.	Ops.	Insts.	Ops.	Insts.	Ops.
Vector Int	V_LOGIC	0.01	0.13	0.03	0.35	0.59	2.71
	V_SHIFT	0.00	0.03	0.07	0.98	0.19	0.79
	V_INTAD	0.01	0.15	0.02	0.34	0.32	1.29
Vector FP	V_FPADD	0.13	0.61	1.71	12.37	4.96	18.80
	V_FPMUL	0.17	0.59	1.39	10.47	5.16	19.44
	V_RECIP	0.03	0.06	0.00	0.05	0.26	1.04
Vector Mem	V_LD	0.31	5.47	2.64	19.70	7.63	29.77
	V_ST	0.17	3.02	1.40	11.63	4.01	15.64
	V_GATH	0.00	0.00	0.04	0.20	0.27	1.66
	V_SCAT	0.00	0.00	0.00	0.00	0.13	0.81
Int	S_ADD	5.11	4.69	8.11	4.01	3.10	0.31
	S_LOGIC	5.72	5.12	5.17	2.33	3.07	0.34
	S_SHIFT	0.63	0.59	1.35	0.77	0.28	0.03
	POP_LZC	0.02	0.02	0.08	0.05	0.12	0.01
FP	FP_MUL	6.80	6.24	3.03	1.78	1.31	0.17
	FP_ADD	7.17	6.56	5.22	2.92	1.88	0.20
	RECIPR	0.26	0.23	0.38	0.23	0.09	0.01
Mem	LD	10.30	9.22	5.13	2.55	3.46	0.39
	ST	5.23	4.71	2.86	1.53	1.08	0.09
	BLK_LD	0.90	0.84*	0.17	0.09*	0.03	0.00*
	BLK_ST	0.62	0.57*	0.28	0.12*	0.37	0.04*
Addr. Comp.	A_ADD	8.37	7.48	15.04	6.64	17.26	1.78
	A_MUL	0.16	0.14	0.91	0.35	0.41	0.04
Branches	BR	6.66	5.94	3.68	1.52	3.27	0.34
Misc.	MOV	12.14	11.00	14.12	5.88	21.85	2.23
	A_SPILL	7.31	6.64	9.66	4.64	9.94	1.08
	S_SPILL	21.75	19.93	17.49	8.51	8.82	0.94

Table 4. The Proportion of *Instructions* and *Operations* of Various Types in the Three Subclasses of Benchmarks

the A unit to the S unit to carry out these functions. The number of MOV instructions could be reduced significantly if the A unit has a more complete computation capability.

Block Loads and Stores (BLK_LD and BLK_ST)

The BLK_LD and BLK_ST instructions are used to transfer a block of words (up to 64 words long) between the backup B/T registers and memory. The number of words transferred is determined at runtime by the contents of a general-purpose scalar register specified in the instruction. A common use of these instructions is in the saving and restoring of registers during subroutine calls/returns. For scalar programs, subroutine calls are frequent (as will be shown later), and hence BLK_LDs/BLK_STs are used quite frequently. BLK_ST is also used as a mini CMR (Complete Memory References) instruction which blocks further instruction issue until all outstanding memory references of the CPU are completed. (The CRAY Y-MP requires memory

* BLK_LD and BLK_ST instructions move a block of words (up to 64 words long) between the scalar backup registers and memory; the number of words transferred is determined at runtime by the contents of a general-purpose scalar register. Since we do not simulate program execution, and since the HPM does not monitor general-purpose registers, we do not have access to this value. Hence we are unable to expand each BLK_LD/BLK_ST instruction into the equivalent multiple operations it executes.

overlap hazards between block reads and block writes of memory to be detected in software, and the CMR instruction can be used to ensure sequentiality of such memory references.) Hence BLK_STs are much more frequent than BLK_LDs for the vector programs.

Floating-Point Operations

The scalar benchmarks have an equal number of floating-point additions and multiplications. The vector benchmarks also have equal numbers of these operations, with the difference that almost all of them are executed by vector instructions. We see a fairly good balance of these operations again in the moderately-vector benchmarks. Here we notice that a large fraction of the floating-point operations are vectorized. Vector floating-point operations constitute 23% while scalar floating-point operations constitute only about 5% of all operations for the moderately-vector benchmarks. The scalar and vector portions of the CRAY Y-MP use common floating-point ADD and MULTIPLY functional units. We observe that only the moderately-vector benchmarks have any significant mix of scalar and vector floating-point operations that might result in conflict for these shared functional units.

Division on the CRAY Y-MP is implemented using multiplication and reciprocal approximation. We observe that there are very few reciprocal instructions (either RECIP or V_RECIP), and hence there are very few division operations in the programs.

Memory Operations

Across the benchmark subclasses, memory load operations (scalar and vector operations together) are roughly twice as frequent as memory store operations. This justifies the presence of two memory load ports and one memory store port in the processor. We note that the ratio of loads to stores is not necessarily a result of having two-operand instructions in the architecture. The issue is complicated by the reuse of temporary results in registers.

From Table 4, we observe that memory operations are the single most frequent class of operations for the vector and moderately-vector benchmarks, and they are second only to the register-register move operations in the scalar benchmarks. Considering that memory access is not a short latency operation, this justifies the extra attention paid to the memory system in the CRAY X-MP and the CRAY Y-MP (the CRAY-1 had a single memory port, while the X-MP and the Y-MP have three data memory ports). We note from Table 2 that for the non-scalar benchmarks, usually more than 90% of the memory operations are executed in vector mode (V_LD and V_ST). When executed in vector mode, the memory latency for an individual operation is hidden by the pipelined nature of operations, and this is significant for performance. Therefore the machine has less need to rely on a data cache for fast memory accesses. Vector scatter/gather instructions (V_GATH and V_SCAT), which transfer data from a set of memory locations specified in a vector register, are used quite infrequently even in the highly-vectorized benchmarks. The need for these instructions is, however, dependent upon the nature of programs.

Address Computation

Most of the address computation instructions are additions (A_ADDs). Address computation instructions are used, for example, to add an index to a base register. Address computation instructions are also commonly used in the CRAY Y-MP for incrementing the loop counter. Address multiplication operations are infrequent.

Branch Instructions

As expected, branch instructions are most frequent in scalar code. However, the branch frequency in Table 4 is much less than the usual 20% of all instructions or so reported for general-purpose programs [8,9,12]. A significant reason for this is the fact that the compiler unrolls loops. First, this eliminates several loop-control branches. Second, loop-unrolling results in the compiler generating several spill instructions because of the increased pressure on the primary registers. These instructions are not present in other architectures, and they decrease the proportion of branch instructions on the CRAY Y-MP. We also note that scientific code inherently has fewer branches than non-scientific code. We discuss the frequency of branches in more detail in the section on basic blocks. In addition to the frequency of branches, the nature of the branches is important to machine efficiency in executing programs. Unconditional branches, for example, do not have to cause bubbles in the pipeline since the branch destination is known at compile time itself. Table 5 provides categorization of the branches into different varieties.

Benchmark Subclass	Branch Type			
	Uncond.	Subroutine Calls	Conditionals	
			Loop Ctrl.	Data-Dep.
Scalar	7.4	24.8	19.6	48.2
Moderate	1.8	7.6	58.5	32.1
Vector	0.6	10.0	60.5	28.9

Table 5. Percentage of Branches of Various Types

Unconditional branches form a non-negligible 7.4% of all branch instructions for scalar code; their proportion is much less in the other two benchmark subclasses. Subroutine calls are implemented in the CRAY Y-MP by a special branch instruction that saves the current program counter (PC) at a specific location and branches to the subroutine. Having a large number of subroutine calls can result in code that is less vectorizable. For example, loops with subroutine calls are usually not vectorizable (except for some vector intrinsic function calls where a *vector* can be passed as an argument to the function and then the function is executed in vector mode). The data bear this out: close to 25% of the branches in the scalar programs are subroutine calls, while their proportion is around 10% of all branches for the other two benchmark sets. We also note from Table 4 that branches themselves are less frequent in the vector benchmarks.

Conditional branches are the most frequent of all branches, across all programs. Although conditional branches are detrimental to performance, the more predictable loop-control conditional branches can be handled efficiently. Conditional branches in the CRAY machines are decided based on the contents of a register; the register used could be either A0 or S0. Usually the compiler uses conditional branching based on A0 for loop-control branches, since loop counters are maintained and incremented in the A unit. Conditional branching based on S0 is used for implementing data-dependent branches, such as if-then-else constructs. Table 5 splits conditional branches into the above two classes. The data indicate that 50% of all branches in the scalar benchmarks are data-dependent conditional branches. Also, data-dependent branches are about two-and-one-half times as frequent as loop control branches.

Given that scientific code is dominated by loops, one can expect most of these branches to occur within loop bodies. Loops with data-dependent branches within them are likely to be scalar; it is thus natural to find scalar code having a significant fraction of these branches.

We observe that for the moderately-vector and vector benchmarks the proportion of loop-control branches has gone up, indicating fewer data-dependent branches per loop. This is a good reason why these benchmarks are more vectorizable than the scalar set.

4. BASIC BLOCKS

A *basic block* is defined to be a straight-line fragment of code with a single entry point (the first instruction) and a single exit point (the last instruction). Once program control enters a basic block, all the instructions in it are executed. The entry point could be the start of a program or either the destination or fall-through location of a branch; the exit point is either a branch or an instruction preceding the destination of a branch (since the destination of a branch is the start of a new basic block).

The nature and sizes of basic blocks play an important role in determining program performance, because several compiler optimizations (such as local register assignment and code-scheduling) are conducted within basic block boundaries unless the hardware or the compiler supports speculative execution of instructions that lie beyond as-yet-unexecuted branches. Larger basic blocks provide better opportunities for effective code scheduling. Folklore has it that basic blocks are small — papers in the literature report average branch instruction frequencies of 15% to 20%, and thus small basic blocks, in general-purpose programs [8,9,12]. In addition to the size of the average basic block, the distribution of basic blocks with respect to their sizes is important, since if both small and large blocks exist in significant numbers, the compiler could incorporate different techniques to tackle the two varieties of basic blocks.

Figure 1 presents the cumulative (dynamic) distribution of basic blocks in the three subclasses of our benchmark set. The solid lines in the figure present the cumulative frequencies of basic blocks of various sizes (in instructions), for each of the benchmark subclasses. The dotted lines present the cumulative contribution to program operations of basic blocks of various sizes. The general shape of the three solid curves indicates that basic blocks range in size from one instruction to beyond a 100 instructions for all the benchmark subclasses. The basic blocks are distributed across the entire range, instead of being clumped near the average block size. 90% of the basic blocks of vector programs are spread over sizes from one to 64 instructions. About 8% of the blocks of vector programs are larger than 80 instructions in size. For moderately-vector programs, 90% of the blocks are spread over sizes from 1 to 45 instructions. For scalar programs however, 90% of the blocks are shorter than 21 instructions. Scalar programs, with more frequent branches, expectedly have smaller basic blocks.

Blocks that are larger than 125 instructions are non-negligible in number, for all three program subclasses: they form about 3% of the instructions for vector programs, about 2% for the moderately-vector programs, and about 1.5% for the scalar programs. The scalar programs have a large number of single-instruction blocks (about 11%). Surprisingly, the vector programs also have about 9% single-instruction blocks. (The scalar code in vector programs, used to set up work for the vector instructions, contain small blocks, as in scalar programs.) The median[7] block size is about 14 instructions for vector programs, about 18 instructions for the moderately-vector programs, and between 8 and 9 instructions for the scalar programs.

The sizes of the basic blocks in our benchmarks are distributed over a wide range, as can be noticed from the cumulative frequency distributions in Figure 1. For example, we observe a significant number of blocks less than 10 instructions in size, and a significant number larger than 125 instructions in size. This indicates that both small and large blocks are important on the CRAY Y-MP. Two things are significant with regard to basic blocks on the CRAY Y-MP. One, we are studying code vectorized by a production compiler; vector instructions execute the equivalent of several

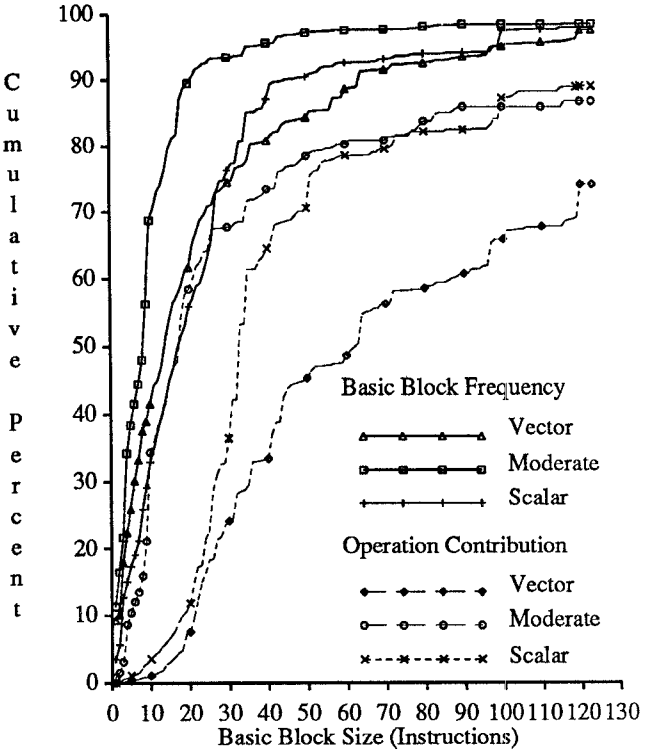

Figure 1. Basic Blocks in the Benchmarks

scalar instructions. Most of the studies reported are of branches in general-purpose code that have only scalar instructions. Two, we notice large basic blocks in all the benchmarks, including code where only a small portion of the operations are vectorized. We discuss below the reasons for such a distribution of basic blocks on the Y-MP.

In addition to the nature of the application, the nature of the compiler and the machine play a significant role in determining the size of the basic blocks. A major reason for the large basic blocks is that the compiler unrolls loops to exploit parallelism, and to tolerate the long latencies of the CRAY Y-MP, eliminating several[8] loop branches in the

[7]Due to the large variance in the data, the arithmetic mean is somewhat undesirable in characterizing the data. We prefer to use the median instead. This is where the graph crosses the 50% mark on the y-axis.

[8]The exact amount of loop unrolling, and hence the number of branches eliminated, is dependent on the size of the original loop body since the compiler unrolls loops up to the size of the instruction buffer (1K bytes).

process and increasing basic block sizes. Second, the compiler also generates in-line code for, or expands in-line, several small subroutines, again eliminating branches and increasing basic block sizes. Third, the presence of a Vector Mask (VM) register in the CRAY Y-MP processor enables the compiler to vectorize loops that have conditional branches in their bodies. The VM register is used to discard the results of operations of the vector instruction that would not have been executed, due to control flow, in a scalar version of the code. Similarly, the presence of a scalar merge instruction enables elimination of several scalar conditional branches. Fourth, the spill and move instructions generated by the compiler account for a very significant fraction of the instructions generated, as we saw in the previous section. These instructions increase basic block sizes. Finally, the scalar portion of the CRAY machines use simple instructions, and hence naturally need more instructions to carry out the job.

Thus large basic blocks are present, in fair proportion, in all three subclasses of programs. However, if most of the program execution time is spent in small basic blocks, the presence of large basic blocks would be immaterial to program performance. For example, if most of the vector instructions were present in the smaller basic blocks, then one would expect a very significant portion of the program execution time to be spent in these small basic blocks. However, our measurements show that to be not the case.

The dotted lines in the figure indicate the cumulative proportion of program *operations* contributed by basic blocks of various instruction sizes. We notice that the large basic blocks of all three benchmark sets contribute heavily to the overall operation count. It is interesting to note that even basic blocks larger than a 100 instructions in size contribute significantly to the dynamic operation count. Blocks larger than 125 instructions contribute 25% of all operations for vector programs, 11% of all operations for moderately-vector programs, and 13% of all operations for scalar programs. The median of operation contributions is about 18 instructions for scalars, about 33 instructions for the moderately-vector programs, and about 63 instructions for the vectors. These sizes are much higher than the medians for instruction contributions. One can thus expect a significant fraction of the program's execution time to be spent in the large basic blocks. We still notice, however, that as we move from vector to scalar code, the blocks of smaller sizes increase their operation contributions. This is due to the presence of fewer vector instructions in scalar code.

5. INSTRUCTION ISSUE

Parallel instruction issue is the focus of much current research (for example, work on superscalar and VLIW architectures). Parallelism in the issue stage and pipelining of the processor are roughly equivalent in exploiting fine-grain parallelism [10, 17]. A processor with deeply pipelined functional units has less need for parallelism in the issue stage [17]. In this section we investigate the utilization of the issue stage of the deeply-pipelined CRAY Y-MP processor by the benchmark programs.

Table 6 presents data collected by the HPM about the utilization of the instruction issue stage of the CRAY Y-MP. The second column in the table presents the instruction issue rate (instructions issued per clock cycle). Recall that the CRAY Y-MP instructions occur in three sizes: one-parcel, two-parcel, and three-parcel instructions, where a parcel is 16 bits long. The one-parcel instructions are issued in a single clock cycle, while the others take two clock cycles to issue: the first parcel is issued in the first clock, and the rest are issued in the second clock. Table 6 identifies the

Program	Instruction Issue Rate	Issue Stage Utilization		
		by 1st parcel	by 2nd/3rd parcels	by all parcels
MG3D	0.104	0.104	0.022	0.126
SPEC77	0.176	0.176	0.045	0.221
ARC3D	0.212	0.212	0.056	0.268
MDG	0.197	0.197	0.075	0.272
FLO52	0.220	0.220	0.066	0.286
BDNA	0.193	0.193	0.118	0.311
DYFESM	0.335	0.335	0.124	0.459
TRFD	0.386	0.386	0.078	0.464
ADM	0.355	0.355	0.139	0.494
OCEAN	0.423	0.423	0.121	0.544
SPICE	0.264	0.264	0.199	0.463
TRACK	0.291	0.291	0.200	0.491
QCD	0.390	0.390	0.217	0.607

Table 6. Instruction Issue Rate and Issue Stage Utilization

percentage of program execution time for which the instruction issue stage is busy issuing the first parcel (column 3) and the second/third parcels (column 4) of instructions. The fraction of program time spent issuing the first parcel is the same as the instruction issue rate, of course. The last column of the table identifies the fraction of time the issue stage is busy overall (this is the sum of columns 3 and 4).

We first observe that the highest issue stage utilization is 42.3%, for OCEAN. Since the utilization of the issue stage is not very high (on the average), we can say that the issue stage is not a bottleneck during program execution. This is of course due to the fact that the deep pipelines exploit the parallelism in the programs even with the small issue rate. For example, the floating-point add unit has 7 pipeline stages; thus, after issuing a floating-point add, 6 other instructions can be issued before the floating-point add completes. Other factors, discussed later in this section, are also responsible for the low utilization of the instruction issue stage. We note, however, that there may be phases during program execution when the amount of parallelism is high and the issue stage is a bottleneck.

As expected, the vector benchmarks have lower issue rates than the other benchmarks. Vector instructions execute several operations, and hence fewer instructions need to be issued for vectorized programs; if we were to consider operation issue rates instead, the numbers would be much higher for the vector benchmarks. However, another important reason for the low issue rate is the CRAY Y-MP hardware organization. There exists more parallelism among the vector instructions of programs than among the scalar instructions, but the CRAY Y-MP has a relatively limited amount of vector resources (vector registers, memory ports, functional units), resulting in instruction issue stalls due to *resource conflicts*. The limited resources result in long instruction issue stall times because each vector instruction reserves all necessary resources for time proportional to the number of operations it executes.

The stalls in instruction issue seen in scalar code are mainly due to data dependences between instructions, since

each instruction holds registers only for time proportional to the latency of the instruction's functional unit. The functional units themselves are pipelined and are not a bottleneck, since they can collectively accept scalar instructions at a much higher rate than the issue stage can issue them. We note that despite having a large proportion of spill instructions, the issue stage is not very highly utilized by scalar code. This indicates that the spill instructions are likely being issued for free during clock periods which would otherwise have been stalls for data dependences to be resolved. The highly optimizing compiler would in most cases be able to schedule the spills in this manner; we note that in some cases the spill instructions could be on the critical path and hence may not be executed for free. We also note that although the issue rate is less than 60% on the whole, considering the highly pipelined functional units in the CRAY Y-MP and the long latency for memory, the utilization of the issue stage is quite high.

The fraction of time spent issuing the second and third parcels is small for most of the non-scalar benchmarks. The scalar benchmarks, however, keep the issue stage busy for an additional 20% of the time to issue second and third parcels of instructions. This is because the two-parcel and three-parcel instructions of the CRAY Y-MP are all scalar memory operations, which are found in high numbers only in scalar code.

In conclusion, the issue stage does not appear to be a bottleneck in the CRAY Y-MP for the PERFECT Club benchmarks. Improving the issue stage utilization and program execution speed can be achieved by increasing the resources in the vector unit (*i.e.*, vector registers, functional units, and possibly memory ports) and by cutting down the latency of scalar operations (*i.e.*, the functional units and memory).

6. CONCLUSIONS

We presented a study of the single processor of the CRAY Y-MP using as benchmarks programs from the PERFECT Club set. We characterized the processor by presenting the instruction usage pattern for our benchmarks. We observed, among other things, that on the CRAY Y-MP the vectorized fraction of the dynamic operation count ranges from 4% to 96% for our benchmarks. The average fraction of all operations vectorized is 62% for our benchmark set. Instructions that move values between the scalar registers and their backup registers constitute a significant fraction of all the instructions executed. Very large basic blocks (greater than a 100 instructions in size) are significant in number for the benchmarks studied. Furthermore, both small and large basic blocks contribute significantly to the operation count of the programs. Thus, it is worthwhile to concentrate on speeding up both small and large basic blocks. The instruction issue rate is less than 0.5 instructions per cycle for our benchmarks, leading us to believe that the issue stage does not appear to be a bottleneck during program execution.

Acknowledgements

The work of the first two authors was supported in part by NSF Grant CCR-8919635. The first two authors would like to thank the Future Architecture Specification Team (FAST) of Cray Research, Inc., Chippewa Falls, especially Chris Hsiung, for their assistance and their support of this research. Part of this work was carried out while the first author was a summer student at Cray Research, Inc. The first author would also like to thank William Taylor for his help with the benchmarks. Finally, thanks to referee A for detailed comments.

References

[1] "The CRAY X-MP Series of Computer Systems," *Cray Research Inc., Publication No. MP-2101*, 1984.

[2] "The CRAY Y-MP Series of Computer Systems," *Cray Research Inc., Publication No. CCMP-0301*, February 1988.

[3] T. L. Adams and R. E. Zimmerman, "An Analysis of 8086 Instruction Set Usage in MS DOS Programs.," in *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, April 1989.

[4] D. W. Clark, P. J. Bannon, and J. B. Keller, "Measuring VAX 8800 Performance with a Histogram Hardware Monitor," in *Proc. 15th Annual Symposium on Computer Architecture*, Honolulu, Hawaii, 1988.

[5] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, "Supercomputer Performance Evaluation and the Perfect Benchmarks," in *CSRD Report No. 965*, University of Illinois, March 1990.

[6] J. S. Emer and D. W. Clark, "A Characterization of Processor Performance in the VAX-11/780," in *Proc. 11th Annual Symposium on Computer Architecture*, Ann Arbor, MI, 1984.

[7] T. R. Gross, et al, "Measurement and Evaluation of the MIPS Architecture and Processor," *ACM Transactions on Computer Systems*, August 1988.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann Publishers, Inc.

[9] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing Software and Hardware Schemes for Reducing the Cost of Branches," in *Proc. 16th International Symposium on Computer Architecture*, Jerusalem, Israel, June 1989.

[10] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," in *ASPLOS-III*, Boston, MA, April 1989.

[11] J. Kohn, "JUMPTRACE," *Cray Research Inc. Report*, April 1989.

[12] S. McFarling and J. Hennessy, "Reducing the Cost of Branches," in *Proc. 13th Annual Symposium on Computer Architecture*, Tokyo, Japan, June 1986.

[13] F. H. McMahon, *The Livermore FORTRAN Kernels: A Computer Test of the Numerical Performance Range.* Research Report: Lawrence Livermore Laboratories, December 1986.

[14] C. L. Mitchell and M. J. Flynn, "A Workbench for Computer Architects," *IEEE Design and Test of Computers*, Feburary 1988.

[15] J. Rubinstein and D. MacGregor, "A Performance Analysis of MC68020-based Systems," *IEEE Micro*, December 1985.

[16] J. E. Smith and J. R. Goodman, "A Study of Instruction Cache Organizations and Replacement Policies," *Proc. 10th Annual Symposium on Computer Architecture*, June 1983.

[17] G. S. Sohi and S. Vajapeyam, "Tradeoffs in Instruction Format Design For Horizontal Architectures," in *ASPLOS-III*, Boston, MA, April 1989.