README

FYI: The Title Page, Abstract, Acknowledgements, Table of Contents, List of Tables, and List of Figures are printed *after* the Chapters and the References.

Please move them to before the Chapters.

Chapter 1

INTRODUCTION

The supercomputers of a particular generation of machines are the computers that provide the highest performance of that generation. Of the programs that require supercomputer performance, a large subset has been and will continue to be large-scale scientific and engineering applications. In this dissertation, we are interested in the design of current supercomputers and in the characteristics of the large-scale scientific and engineering programs that form almost the entire target workload of current supercomputers.

The current nature of research in computer architecture as well as the current realities of the computer marketplace are such that machine design is, for a significant part, driven by the characteristics of the target workload, rather than the workload being subject to massive restructuring so as to execute well on new machine designs. Powerful high-level language compilers are as yet limited to restructuring programs at only a very low level in the hierarchy of program design; algorithm-level changes, or for that matter effective automatic restructuring of very coarse-grain parallelism in the programs, is well beyond the foreseeable powers of compilers. Furthermore, the effectiveness of even the low-level restructuring done by current state-of-the-art compilers often falls much short of the effectiveness achievable via manual restructuring. For example, we compare the performance of a state-of-the-art vectorizing compiler on a benchmark suite with excellent manual restructuring of the suite, and find the latter to be significantly more effective. Thus, machine design targeted at workload characteristics will be more effective than designs that rely on massive automatic restructuring of programs for high-speed execution. Understanding the behavior of machines while executing target workloads, and understanding the characteristics of the workloads themselves, is hence an integral part of the machine design process today. Such understanding enables the design of machines that have features targeted at workload characteristics, and consequently machines that provide improved performance.

Experimental studies that provide such understanding of machine behavior and workload characteristics of several von-Neumann architectures have been reported in the literature. Studies of a non-vector CISC architecture, [Emer84, Clark88] previous generation microprocessors[Adams89, Rubins85], and current generation "killer micro" RISC microprocessor architecture, [Gross88] have been reported to date. However, no similar study of a vector processor has been reported, although studies of just the vector aspects of vector machines — carried out using only short kernels as workloads — have been reported[Tang88]. Furthermore, from the point of view of workload classes, all the studies reported are of "general-purpose" workloads or some mixture of "general-purpose" programs and a few kernels from "scientific" programs. Workloads consisting entirely of scientific programs have thus not been studied. This dissertation fills this current void in the understanding of machine behavior and of workload characteristics by reporting a study of several aspects of a state-of-the-art vector processor executing complete scientific application programs. Specifically, we study a single processor of the CRAY Y-MP[CRI88], and a benchmark set of scientific/engineering application programs, the PERFECT Club benchmark suite[Cybenk90].

1.1. WHY DO WE STUDY A VECTOR PROCESSOR?

Scientific and engineering applications (together referred to as scientific applications henceforth) usually operate on large quantities of data that are well-structured as single-/multi-dimensional arrays or vectors. Program operations on these data are also usually well-structured; for example, every data element might be manipulated in precisely the same manner in a program segment. The result is a large amount of well-structured parallelism among the operations in scientific programs. Often, however, significant portions of a scientific program does not contain such well-structured parallelism, as will be illustrated in this dissertation. In general, the nature of parallelism in scientific programs and in non-scientific programs varies. Several different machine architectures have been proposed for exploiting different kinds of parallelism.

Parallelism in programs is classified into *coarse-grain* and *fine-grain* parallelism[Gajski85]. Coarse-grain parallelism is parallelism at the process level: a program could consist of several logical subtasks that could largely be executed in parallel asynchronously, except for some amount of coordination among the subtasks. Fine-grain parallelism refers to parallelism at the instruction level within a single process or subtask. Fine-grain parallelism can be further classified into *regular* and *irregular* parallelism. Parallelism is said to be regular when a particular operation has to be carried out on several independent data elements (such as the elements of an array). This is also referred to as *data parallelism* in the literature[Hillis86]. Irregular fine-grain parallelism exists when several different kinds of instructions, each of which operates on a single data element, can be executed in parallel.

Several architectural models for exploiting parallelism have evolved from the von-Neumann model of uniprocessor architecture. Shared-memory multiprocessing and distributed computing are the two architectural models popular for exploiting coarse-grain parallelism. Vector processing and array processing (including such processing as in the Connection Machine [Hillis85]) exploit regular fine-grain parallelism via pipelining and large-scale synchronous parallel operation. Pipelining, multiple functional units, and multiple instruction-issue per clock cycle are commonly used techniques for exploiting irregular fine-grain parallelism within a single processor.

The dataflow model[Trelea82], an architectural model that avoids the von-Neumann model altogether in an attempt to avoid the associated limitations, exploits parallelism of all kinds. Dataflow is an elegant architectural model, but efficient implementations of dataflow architectures are still a topic of research. We do not consider dataflow architectures further in this dissertation.

Since a large fraction of scientific code is either dominated by or has a large amount of regular fine-grain parallelism, vector processing and array processing are possibly the most desirable architectures, from among these alternatives, for processing such code. Until recently, vector processing [Thornt70, CRI76, CRI85, CRI84, CRI88, CDC81, Eoyang88, Watana87, Miura83] has dominated the supercomputer arena; in particular, the Cray series of machines and their predecessor CDC machines have been the fastest supercomputers for over two decades. Other than vector processing, massively parallel processing has also been used for scientific codes (for example [Batche80]). However, the main drawback of massively parallel processing (including that of systolic processing) is the poor performance on the non-data-parallel portions of scientific code. More recently, the massively-parallel SIMD Connection Machine [Hillis85] has made some headway in providing improved performance on general scientific codes by *adapting* many algorithms to this mode of computation[Hillis86]. On the high-performance single-processor front, architectures that can issue more than one instruction per clock cycle such as VLIW machines (for example [Fisher83, Fisher87, Rau88, Charle81] etc.), decoupled access/execute architectures (for example [Smith82, Goodma85, Pleszk86] etc.), and superscalar architectures (for example [Patt85, Oehler90] etc.) have been proposed for scientific code.

Thus, numerous alternative architectures currently exist for executing scientific code. What are the motivating factors for and benefits of studying a vector machine from among the alternatives? Our goal is to understand the characteristics of scientific workloads as well as the effectiveness of machine features targeted at such workloads. It is important to observe in this context that, as mentioned before, scientific code has scalar portions, although a large part of scientific code has data-parallelism. It is relatively straightforward to speedup the execution of data-parallel code by using additional parallel hardware resources. On the other hand, ways of speeding up the execution of scalar code are much less obvious. Although the scalar code is usually considered to be a small fraction of scientific code, it ultimately limits the maximum speedup of the workload achievable via parallelization (Amdahl's Law). Furthermore, we will demonstrate in this dissertation that the fraction itself is often quite significant. Thus, when studying scientific applications, we would be interested in identifying the fraction of typical scientific programs that is scalar, and then in characterizing the data-parallel and the scalar portions separately. Such separation of vector and scalar code will facilitate the design of machine features targeted at the two individual code types.

A vector machine provides an excellent environment for pursuing such goals, since the data-parallel portion of code is executed by *vector instructions* in such a machine, while the scalar portion is executed by *scalar instructions*. A natural separation of scalar and data-parallel code is hence available on a vector machine. Furthermore, the compiler carries out a large part of the separation of code into the fine-grain data-parallel portions and the scalar portions. The amount of such separation achieved is crucial to the performance delivered by the system. Studying a vector system that is equipped with a state-of-the-art *vectorizing* compiler provides us with the additional advantage that the amount of such separation seen in the system is a reflection of the current state of the art.

A study of a vector architecture and of workloads executing on such an architecture is of interest and can be used to draw more general conclusions only if the performance of the vector architecture is at least comparable to or better than that of the alternative architectures. In chapter 2, we provide an overview of the various architectural models for exploiting fine-grain parallelism mentioned above, comparing their costs and performance benefits. Then we discuss the execution of code on a vector machine and explain why we think it is currently a good model for executing scientific code and why it is a good choice for a study such as the one done in this dissertation. Specifically among the alternatives, superscalar and VLIW architectures are often considered as alternatives to vector processing. These architectures can also exploit data-parallelism, although with a different model for specifying the computation, and with different associated costs. We provide a comparative discussion of these machines and vector machines in chapter 2.

1.2. CONTRIBUTIONS OF THE DISSERTATION

As mentioned earlier, the study reported in this dissertation is the first such instruction-level study of a vector machine, carried out using complete scientific applications compiled by a state-of-the-art vectorizing compiler as benchmarks. We explore several aspects of vector machines in our studies. We list below the key contributions of the study.

(1) We study the dynamic program usage of and the effectiveness of various architectural and implementation features of the CRAY Y-MP processor, as well as several program characteristics that are of importance to machine design. The study reflects the effects of a state-of-the-art vectorizing compiler on programs and on machine utilization.

(2) We compare the characteristics of a hand-optimized version of our benchmarks, optimized by a Cray Research team of programmers, that won the 1990 Gordon-Bell PERFECT Award, with the characteristics of the compiler-optimized version. The effect of hand-optimization on instruction-level

program characteristics has not been studied to date.

(3) We separate out and study the characteristics of scalar code in the CRAY Y-MP vector system. We discuss the impact of the characteristics of such scalar code on machine design. The scalar code studied is one of two versions: code not vectorized by a state-of-the-art compiler, and code not vectorized by excellent hand-optimizations. Scalar code ultimately limits the speedup achievable via exploiting parallelism (Amdahl's Law), and hence it is important to execute such code well. No identification or study of this type of scalar code has been reported to date in the literature.

1.3. OVERVIEW OF THE STUDIES

Benchmarking is often used to compare various machines as well as to judge the benefits of various architectural features. However, understanding the *behavior* of benchmarks on existing machines is more useful in evolutionary machine design. Such understanding enables one to identify, for example, bottlenecks in current systems, potential bottlenecks in future systems, frequently-occurring code sequences which should be exploited, little-used machine features that could be removed, etc. Beyond such evolutionary design, program understanding could also trigger significantly new and different architectural ideas. A study of benchmark behavior is usually more informative and useful when carried out a low level, i.e., the programs have to be studied at a level close to machine code, rather than at the high-level-language source-code level. Our study is such a detailed instruction level study of a vector machine, addressing several issues with regard to program characteristics and machine behavior. However, the studies are not all-encompassing, due to this being the first such study of a vector machine, and also partly due to the relatively limited access available to such machines.

In the chapter on study methodology (chapter 3) we provide an overview of the important issues to be studied in a vector machine, and discuss the CRAY Y-MP processor, the benchmarks, and our study methodology. Chapters 4 and 5 present studies of various issues addressed in this dissertation, as described below.

The studies reported in this dissertation pertain to a single processor of the vector machine; we focus on processor design in our studies. A significant fraction of the study is directed at instruction mixes in the benchmarks. The frequency of various instruction types is crucial to the design of the functional unit architecture (including memory architecture). We discuss the various instruction types in detail individually. For example, we focus on branches since branch-execution is crucial to fast program execution, especially for scalar code. Further, since vector machines execute vector instructions that execute several *operations* each, we also expand vector instructions into operations and measure operation frequencies. These operation frequencies provide a better picture of machine utilization than instruction counts. We study the sizes of basic blocks in the benchmarks, since these sizes are crucial to the kind of code optimizations and scheduling techniques the compiler can employ. Instruction issue rates and operation issue rates are the subject of much current research, and we investigate them for our machine. All the above studies are presented in chapter 4.

Hand optimization is still often critical to program performance on supercomputers today. In order to compare the performance of the compiler with that of hand-optimization, and to contrast the characteristics of programs generated by the two, we also study a hand-optimized versions of our benchmarks. This version was hand-optimized by a team of Cray Research, Inc. programmers and won the 1990 Gordon-Bell PERFECT Award for the fastest version of the PERFECT Club benchmarks. We discuss the issue of hand optimization in more detail in the chapter on study methodology (chapter 3). Throughout the study, we compare the hand-optimized and compiler-optimized versions of the benchmarks.

We also separate out the scalar code in our benchmarks, and study several issues that are specific to the fast execution of such codes. We discuss in the methodology section of chapter 3 how scalar code is identified and separated. In chapter 4 we study the instruction mixes and basic block sizes of the scalar basic blocks, in addition to those of the entire programs. Then, chapter 5 is devoted to a detailed study of issues relevant to the scalar code. Among the issues addressed are the data dependencies seen between instructions, fast execution of branches, and the tradeoff of machine pipelining and latencies for fast execution of scalar codes.

The final chapter (chapter 6) of the dissertation provides a summary of the studies carried out, and discusses some possible future directions for the work presented here.

Chapter 2

ARCHITECTURES FOR EXPLOITING FINE-GRAIN PARALLELISM

2.1. INTRODUCTION

The goal of machine designers is to appropriately choose and organize the limited hardware resources available for a machine, so as to decrease program execution time. The view of the machine provided to the compiler by the *organization* of the hardware plays an important role in determining the effectiveness of the compiler at utilizing the hardware. Here we first take a conceptual look at the execution of a program on a single von-Neumann processor, in order to understand the performance and cost implications of various processor organizations that exploit fine-grain or operation-level parallelism in programs.

The execution of the operations of a program is constrained by data-dependencies and control-dependencies amongst the operations. An operation can be executed only after all operations that play a role in producing the operands of this operation are completely executed. And, in a program for a von-Neumann machine, an operation cannot be executed until control flow reaches that operation as a result of all logically-preceding branch operations being fully executed. These constraints give rise to a *partial order* of program operations, which has to be satisfied by any execution of the program.

2.2. PROGRAM EXECUTION ON LIMITED RESOURCES

On a machine with infinite resources, determining the order of execution of operations (i.e., the timetable of operation execution, or, the execution schedule) is trivial: each operation is executed as soon as its constraints are satisfied, *i.e.*, as soon as all operations it is dependent on are executed. Hence, the execution time of the program is determined by the longest sequence (or chain) of constraints in the program. The sum of the latencies of all operations on this chain, the *critical path*, is the program's execution time. For example, consider the execution schedule shown in figure 2.1. The arrows in the figure indicate (data or control) dependencies amongst the operations (the destination operation of the arrow is dependent on the source operation of the arrow). For simplicity, we do not distinguish between data- and control-dependences in the example. Note that the effect of either dependence is the same: an operation that is dependent on other operations cannot be issued until the dependencies are resolved. The operations indicated in bold numbers form the critical path, and their execution time is the execution time of the program. Single-cycle operation latencies are assumed in this example.

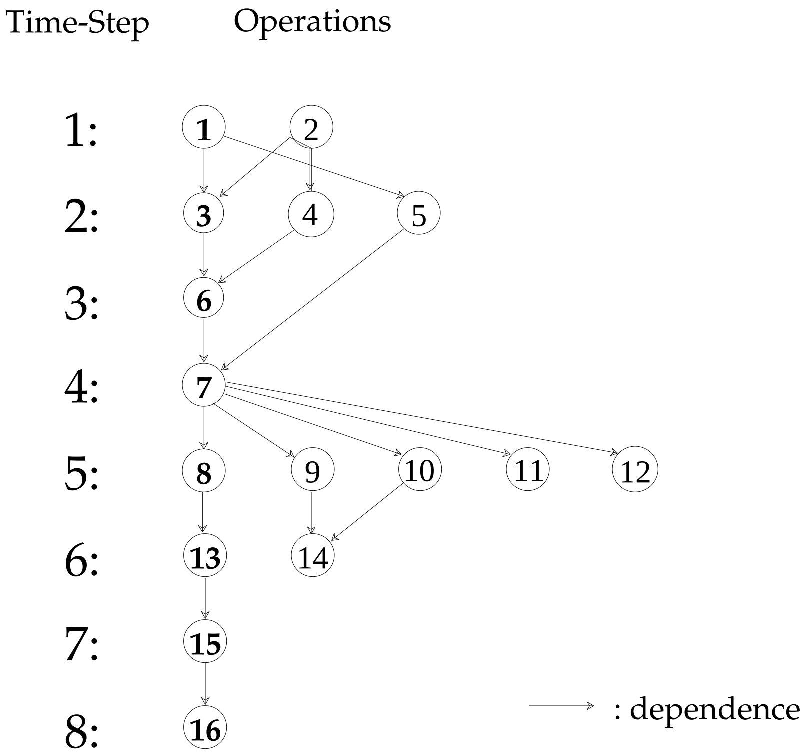


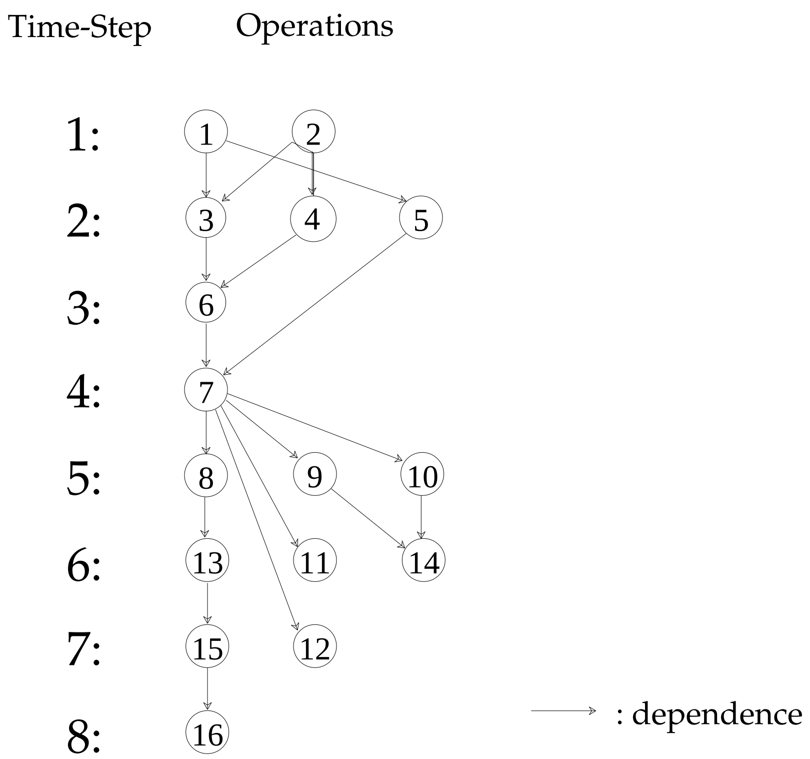
Figure 2.1: Execution on Infinite Resources

Limiting the resources available to execute a program makes the scheduling of operations for execution an NP-complete problem[French82, Garey79, Landsk80]. Consider figure 2.1 again. The schedule can be thought of as a set of operation slots arranged in a rectangle, with several slots containing operations being executed, and others being empty. Each row of slots is executed in a single timestep (clock cycle or period); all the slots of a row are executed simultaneously. For an infinite resource machine, the depth of the rectangle (the time axis) is the length of the critical path of the program. The width of the schedule for such a machine is *unbounded*: at any point in time, the width of the schedule determines the number of operations being executed in parallel, and there is no bound on this number for an infinite-resource machine. Limiting the resources available in the machine now bounds the width of the schedule. Furthermore, it also bounds the quantity of operations of each *type* that can be executing simultaneously. We will discuss these *machine constraints* on the execution schedule in detail shortly. The schedule for an infinite-resource machine will usually violate, at several time points in the schedule, the constraints imposed by a particular real machine because of its limited hardware resources. Operations that violate these constraints have to be relocated into other slots, possibly requiring (in order to satisfy program dependencies) the relocation of operations that do not violate machine constraints in the schedule for the infinite-resource machine. Relocating the operations in the slots to satisfy the constraints of a real machine will invariably require an increase in the depth of the schedule, i.e., an increase in program execution time. The problem of packing operations into this constrained rectangle while trying to minimize the depth of the rectangle and while still satisfying the original data-dependencies and control-flow constraints is NP-complete[1]. We note that several heuristic algorithms that provide reasonably good performance have been developed for the scheduling problem and have been incorporated in state-of-the-art compilers [Landsk80, Ellis85, Hsu85].

From the scheduling point of view, it is easier to achieve high performance on a machine that provides a schedule rectangle that is broader and has fewer machine constraints. However, such a machine usually needs more hardware resources and is hence costlier. In some cases, such a machine is also harder to build — it is hard to achieve short clock cycle time on a machine that has complex hardware. For example, one can build a register file that has more than a handful of ports, but it is very hard to still maintain single-cycle selection of and access to the individual registers without having a relatively-large clock cycle. Real machines hence involve tradeoffs among cost, ease of building, and schedulability (schedulability directly determines achievable performance). Importantly, the desirable optimum schedule rectangle that a real machine should provide is heavily influenced by the nature of the programs that will be run on the machine.

To illustrate the tradeoff for some given machine workload, let us consider the example of figure 2.1 again. A machine that allows a maximum of three operations to execute in parallel at any time performs as well for this program as a machine with infinite resources though the schedule is actually different, as shown by the schedule in figure 2.2. A machine that allows a maximum of only 2 operations to be executing in parallel at any given time is only one clock period slower for this program, as shown by the schedule in figure 2.3. The performance of these two realistic machines

[1] This is similar to the bin-packing problem.



— maximum of 3 operations/cycle

Figure 2.2: Execution on Limited Resources

— maximum of 3 operations/cycle

could however be quite different for some other program.

In this dissertation, the workload of interest is scientific programs. We will discuss machine design only in this context henceforth. Several processor architectures, each imposing its own unique set of machine constraints on the schedule rectangle, have been proposed for such workloads, and are discussed in the next section. We first outline the schedule rectangles provided by each architecture, and compare and contrast their cost and potential performance. Then we focus on the merits and demerits of a particular architecture, the vector LOAD/STORE architecture, and discuss why it is an interesting architecture for the study of scientific workloads. Finally, we concentrate on the design aspects of the vector architecture.

2.3. EXECUTION SCHEDULES ON VARIOUS ARCHITECTURES

The main processor features that determine the nature of the schedule rectangle presented by the processor are: instruction issue capability, the quantity and variety of functional units, memory architecture, and pipelining. We discuss below the effects of

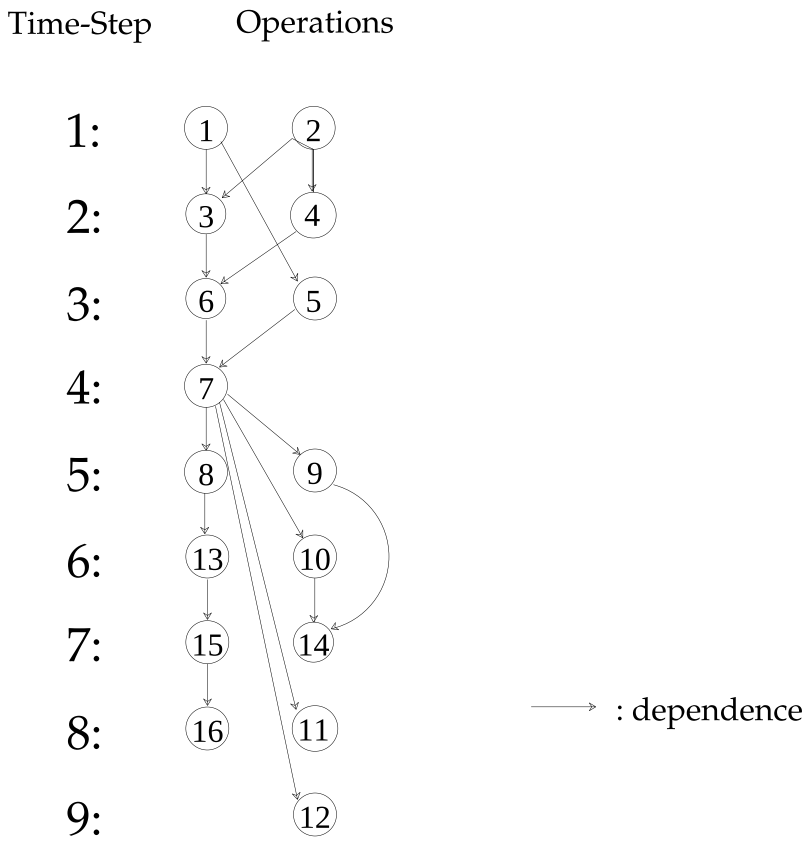


Figure 2.3: Execution on Limited Resources

— maximum of 2 operations/cycle

these features in detail. The register file architecture and the operand/result buses also play a role, albeit a less significant one. We mention their effects briefly.

Let us first consider architectures that issue only one instruction each clock cycle. (In the rest of this chapter, we use the terms instructions and operations interchangeably, except when discussing vector instructions which execute multiple operations. In this case, we explicitly use the term vector instruction.) The most basic processor has a single non-pipelined ALU, and the schedule for such a processor is a simple one-dimensional rectangle where operations execute strictly in sequence. Scheduling for such a machine is trivial.

Suppose the ALU of the above machine is pipelined into 5 stages, and the clock cycle is approximately one-fifth that of the previous machine[1]. Now each stage of the pipeline can be executing an operation, resulting in a maximum width of 5 operations for the schedule rectangle. However, the rectangle is further constrained by the fact that the ALU can only accept one new operation ever clock: each row of the schedule can only be a maximum of 1 operation wider than the previous row. Overall, compared to the previous processor, program execution is much faster due to the possible execution overlap of operations. (Note that the schedule rectangle is just the common pipeline diagram of a single pipeline, the ALU).

Contrast the above processor with one that has 5 non-pipelined ALUs, has the same clock cycle as the baseline non-pipelined single ALU machine, and can issue 5 operations per clock cycle. The rectangle here is a maximum of 5 operations wide, and has no other machine constraints. A code fragment consisting of 5 independent operations is executed by this machine in 1 clock cycle — all the 5 operations can be issued in the same clock. To execute 5 independent operations, the pipelined machine needs 9 clock cycles (5 to initiate the 5 operations + 4 more for the last operation to complete), and is 1.8 times slower than this machine (the pipelined machine's clock is 5 times faster). To execute 50 independent operations, the pipelined machine is only 1.08 times slower than this machine! However, the multiple ALU machine is probably 5 times more expensive due to the replicated hardware and the communication crossbar necessary; pipelining is cheaper than hardware replication. On the other hand, pipelining involves complex control hardware which is harder to build; it might also affect overall functional-unit latencies[Kunkel86].

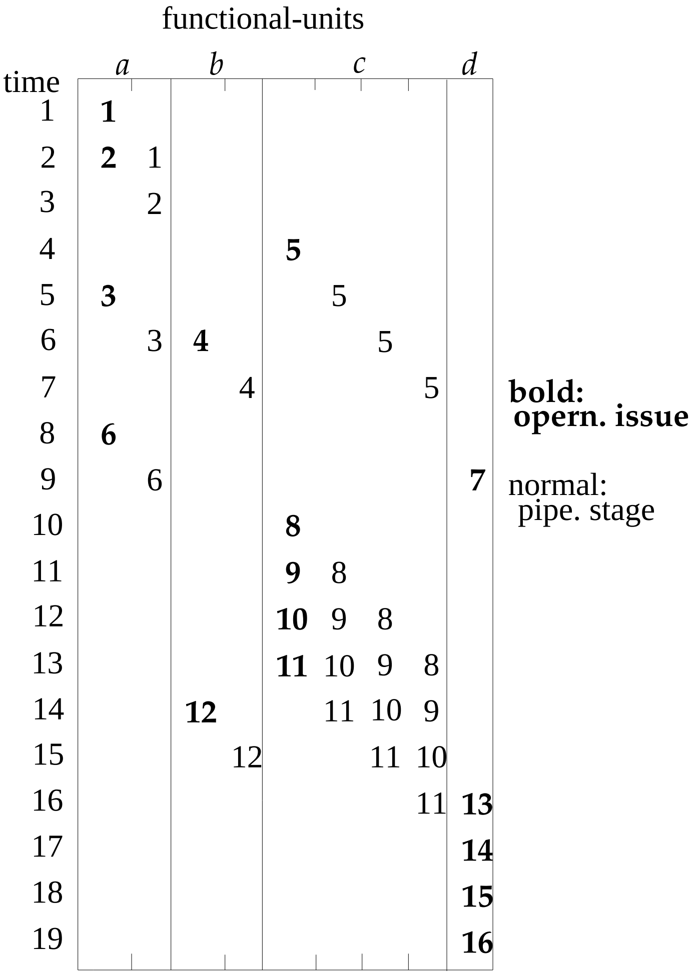
Scheduling in the presence of dependencies is non-trivial for both the above processors. One has to appropriately choose operations for parallel execution so that the overall execution time is minimized.

[1] It is hard to obtain exactly one-fifth the original clock cycle, due to various hardware implementation difficulties associated with pipelining[Kunkel86]. We ignore this issue in the discussion here.

Let us get back to a single instruction issue machine, but one that has independent, dedicated, pipelined functional units instead of a single ALU that executes all *types* of operations. A fundamental advantage of this architecture is that all the different types of operations are not slowed down to the execution time of the slowest operation type. A long latency operation can be issued first, and during its execution several other operations can be issued and possibly short latency operations can even be fully executed. Figure 2.4 shows the execution schedule, on such a machine, of the example code fragment of figure 2.1. We assume four functional units *a, b, c, d* of pipeline depths 2, 2, 4, and 1 stages respectively. The operations of the example code fragment are, as shown in the figure, assumed to be of certain types and issued to the corresponding functional units. An operation is indicated in bold print in the clock cycle in which it is issued; its execution in the rest of the pipeline stages of the functional unit are indicated in normal print. For example, operation 1 is issued in clock cycle 1 and completes execution in clock cycle 2, while operation 2 is issued in clock cycle 2 and completes execution in clock cycle 3. Operation 3 has to wait until operation 2 completes execution, and hence is not issued until clock 5. Observe that short latency operations 3 and 4 are issued and executed during the execution of the long latency operation 5.

The width of the schedule rectangle of this architecture is equal to the sum of the number of pipeline stages in each of the functional units (i.e., the maximum number of operations that can be in execution simultaneously). The width of the example machine is 9 (2+2+4+1) operations. Since only one instruction is issued per cycle, no row of the schedule can be more than 1 operation wider than its preceding row. This further reduces the number of pipeline stages that can be busy, since the first pipeline stage of only one of the functional units can be busy at any time. Thus, the maximum number of functional unit pipeline stages of the CPU that can be busy simultaneously is equal to the number of stages in the deepest functional unit pipeline. Of course, the maximum number of operations of a given type that can be executing simultaneously (*i.e.*, in any row of the schedule) is equal to the number of pipeline stages in the functional unit that executes that operation type. For example, functional unit *c* can have a maximum of 4 operations in execution, as in clock cycle 13. Scheduling in the presence of dependences for such a machine is again non-trivial. A machine that provides such a schedule rectangle is the scalar unit of the CRAY Y-MP [CRI88].

Suppose one were able to issue an instruction to each functional unit of the above machine every clock cycle. The schedule rectangle provided by this machine is different from that provided by the above machine mainly in the following way: each row of the schedule can be *n* wider than the previous row, where *n* is the number of instructions issued per clock cycle (and hence the number of functional units in the machine). Of course, the functional units dictate the number of operations of each type that can be in execution simultaneously. Figure 2.5 shows an execution schedule of the same code fragment on a machine with the same functional units as in figure



functional-units

time	*a*		*b*		*c*				*d*
1	**1**								
2	**2**	1							
3		2							
4					**5**				
5	**3**					5			
6		3	**4**				5		
7				4				5	
8	**6**								
9		6							**7**
10					**8**				
11					**9**	8			
12					**10**	9	8		
13					**11**	10	9	8	
14			**12**			11	10	9	
15				12			11	10	
16								11	**13**
17									**14**
18									**15**
19									**16**

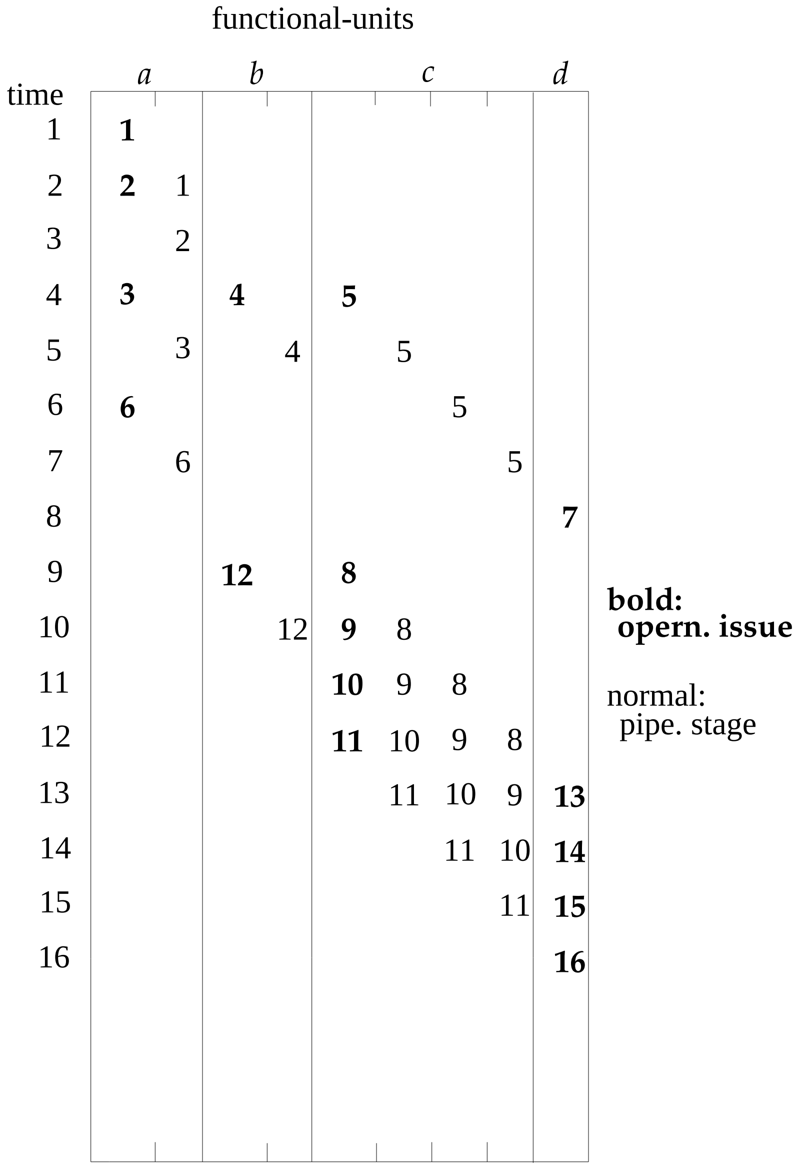
bold:
opern. issue

normal:
pipe. stage

Figure 2.4: Execution on Pipelined Functional Units

— *single operation issue*

2.4, but with a very long instruction word. In clock 4, for example, three operations are issued by this machine. The example code takes 16 clock cycles to execute on this machine, as opposed to the 19 clocks on the previous machine. A requirement for better utilization of such machines is uniform parallelism across all instruction types.



functional-units

time	*a*		*b*		*c*				*d*
1	**1**								
2	**2**	1							
3		2							
4	**3**		**4**		**5**				
5		3		4		5			
6	**6**						5		
7		6						5	
8									**7**
9			**12**		**8**				
10				12	**9**	8			
11					**10**	9	8		
12					**11**	10	9	8	
13						11	10	9	**13**
14							11	10	**14**
15								11	**15**
16									**16**

bold:
opern. issue

normal:
pipe. stage

Figure 2.5: Execution on Pipelined Functional Units

— *very long instruction word*

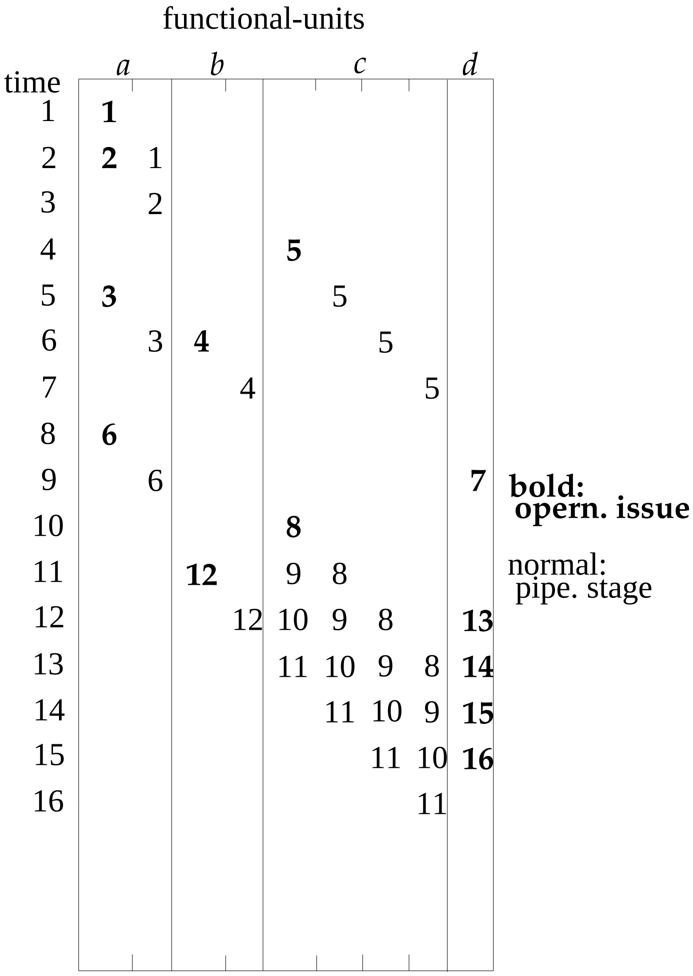
For example, although operations 8 through 11 are independent, they have to be issued in series since they are of the same operation type and the machine is equipped with only one functional unit that can execute that operation type. However, exploiting the parallelism across operation types (as in clock 4 for example) enables this machine to execute the code in fewer clock cycles. VLIW architectures[Fisher83] provide such a schedule rectangle.

Vector machines provide a schedule that lies, in the spectrum of machine constraints, in between the above two schedule rectangles. Although instruction issue is limited to one per clock cycle, each vector instruction that is issued initiates an operation every clock cycle for VL clocks, where VL is the vector length of the instruction. Thus, a restricted form of multiple operation issue is achieved. Figure 2.6 presents the execution schedule of the code fragment on such a vector machine. Operations 8 through 11 of the code are bundled into a vector instruction[1] for this vector machine. For ease of comparison, we represent this vector instruction as instruction 8, which is issued on clock 10. This instruction in turn initiates operations 9, 10, and 11 in the next three clocks. Note that instruction 12 can now be issued on clock 11 since the instruction issue stage is not busy. Similarly, instructions 13 through 16 can be issued earlier. We observe that the vector machine also executes the code in 16 clock cycles, as opposed to the 19 clock cycles taken by the single-issue machine. We note that the execution times achieved on the various machines are highly dependent on the workload.

So far we have considered the effects of the presence of pipelining, the resource architecture, and the instruction-issue capability on the schedule rectangle. We will not discuss in detail the effects of the register file architecture and the operand/result buses. Suffice it to note that the these issues affect performance: for example, for a fixed number of ports per register file, a split register file architecture allows more operands to be read each clock cycle and hence allows more instructions to be issued per cycle. If a register file has to be able to communicate with several functional units that are geographically far apart, the length of the operand/result buses might limit the clock rate.

An important factor, not yet discussed, that affects a schedule rectangle is the *degree* of pipelining of the functional units of the processor. We have thus far only discussed the effect of the mere presence of pipelining. Let us now consider the effect of various *degrees* of pipelining. Consider a machine with two functional units (say an integer unit and a floating-point unit), each having 3 pipeline stages. Consider a program that has 6 independent operations, 3 of each type. If we issue only one

[1] Usually, the vectorization of operations also results in the reduction of index and loop-control operations. For clarity, we ignore these effects here.



time	a		b		c				d
1	**1**								
2	**2**	1							
3		2							
4					**5**				
5	**3**					5			
6		3	**4**				5		
7				4				5	
8	**6**								
9		6							**7**
10					**8**				
11			**12**		9	8			
12				12	10	9	8		**13**
13					11	10	9	8	**14**
14						11	10	9	**15**
15							11	10	**16**
16								11	

Figure 2.6: Execution on Pipelined Functional Units

— single instruction issue, vector machine

instruction per clock, the program will execute in 8 clock cycles (6 to issue all the operations + 2 for the last operation to complete). (Figure 2.7 shows the schedule.) Now suppose the functional units have 6 pipeline stages each, and the clock cycle is

clock	Integer Unit			FP Unit		
1	**i1**					
2	**i2**	i1				
3	**i3**	i2	i1			
4		i3	i2	**f1**		
5			i3	**f2**	f1	
6				**f3**	f2	f1
7					f3	f2
8						f3

Shallow-Pipelined Machine — *single instruction issue*

clock	Integer Unit						FP Unit					
1	**i1**											
2	**i2**	i1										
3	**i3**	i2	i1									
4		i3	i2	i1			**f1**					
5			i3	i2	i1		**f2**	f1				
6				i3	i2	i1	**f3**	f2	f1			
7					i3	i2		f3	f2	f1		
8						i3			f3	f2	f1	
9										f3	f2	f1
10											f3	f2
11												f3

Deep-Pipelined Machine — *single instruction issue*

clock	Integer Unit			FP Unit		
1	**i1**			**f1**		
2	**i2**	i1		**f2**	f1	
3	**i3**	i2	i1	**f3**	f2	f1
4		i3	i2		f3	f2
5			i3			f3

Shallow-Pipelined Machine — *very long instruction word*

Figure 2.7: The Effects of Pipeline Depth and Issue Bandwidth

on Program Execution Time

(all operations are independent)

approximately half the original cycle. Issuing one instruction per cycle, the program will now execute in 11 clock cycles (6 to issue + 5 for the last to complete). Note that operation *f3* is now issued *before* operation *i3* completes execution. Since the clock cycles of this machine are half the original clock cycles, this machine takes only 5.5 clock cycles of the original machine to execute the program. Now consider the machine with 3 stages in each functional unit, but with the capability of issuing an instruction to each functional unit every cycle. The program now executes in 5 clock cycles (3 for issue + 2 for the last two operations to complete). Note that the machine with 6 pipeline stages and single instruction issue was not much slower (5.5 clocks vs. 5 clocks)! The difference between the schedule rectangles of the two machines is the following. The shallow-pipelined machine with dual-issue has a schedule rectangle that is a maximum of 6 operations wide, and each row can be 2 wider than the previous row. The deep pipelined machine with single issue has a schedule rectangle that is also a maximum of 6 operations wide, since only one the two functional units gets an instruction on a clock; each row of the schedule can only be 1 wider than the previous row. Importantly, if we consider the overall schedule of the program for this machine, the schedule is divided into many more rows than that for the shallow-pipelined machine. That is, the time interval at which a new instruction can be initiated is smaller, resulting in more choices for scheduling of operations within the same time-frame. Such a schedule might in fact be more desirable than the one offered by the shallow-pipelined machine. There is thus a tradeoff to be considered between the level of pipelining and instruction issue parallelism. The design choices are again obviously dependent on the nature of the programs run on the machines.

Thus far in this section we have discussed the schedule rectangles provided by various architectures, and pointed out the respective tradeoffs involved in cost and potential performance. The choice of a schedule rectangle is determined by its cost and by its predicted performance on the intended workload. First, the machine need only provide a schedule rectangle that is as spacious as the intended workloads can use. Second, replicating hardware resources such as functional units and instruction issue stages provide better schedule rectangles, but these are expensive options. Such replication could also result in excessive pressure on the register-file interfaces (ports) and operand/result buses of the machine. It could be very difficult to provide short clock periods under such pressure; it is hard to build many-ported register file with a short clock, for example. Deeper pipelining of the functional units, on the other hand, might be able to provide schedule rectangles that perform almost as well, without incurring the same costs. We discuss these issues in the next section.

2.4. CHOICE OF PROCESSOR ARCHITECTURE

Scientific workloads have larger amounts of fine-grain parallelism in general, as compared to general-purpose programs. Usually a large amount of this parallelism in scientific workloads is data parallelism. High-performance (supercomputer) processors have historically been providing extensive amounts of hardware resources to exploit the fine-grain parallelism in scientific workloads. (For example, consider the multiple vector as well as scalar functional units, multiple register sets, multiple memory ports, etc., in a processor of the CRAY Y-MP). However, until the early 1980s, an instruction issue mechanism capable of issuing just one instruction per cycle was considered adequate to exploit fine-grain parallelism. (Note that although vector instructions can issue multiple operations per cycle, they are restricted to exploiting data parallelism.) With significant improvements in the early 1980s in compiler techniques for detecting and enhancing parallelism[Fisher81], VLIW architectures [Fisher83] that could issue an instruction to every functional unit on each clock cycle were proposed for programs in general, unlike vector architectures that are targeted mainly at programs with data-parallelism. VLIW architectures provide a schedule rectangle with the least amount of machine constraints, as discussed in the previous section. Several other processor architectures (decoupled, superscalar) that can issue multiple instructions per cycle but having less issue parallelism than VLIW machines have also been proposed for programs in general irrespective of the vectorizability of the programs.

There is no consensus as yet on the amount of operation level parallelism extractable from general programs, and consequently on the appropriate architecture/organization to be used for exploiting such parallelism[Nicola84, Sohi89, Jouppi89, Smith89, Wall91, Butler91]. However, it is folklore that scientific code has large amounts of regular (data) parallelism and some amount of scalar code (which is either mostly sequential or has some amount of irregular parallelism). Using additional hardware resources to exploit the large amount of regular parallelism is simple and straightforward. Methods of speeding up the rest of scientific code are, however, much less obvious. Since this scalar fraction of the code ultimately limits the overall speedup achievable (Amdahl's Law), it is important to improve its execution speed.

This thesis attempts to provide a characterization of large scientific programs, in order to aid the design of appropriate architectures for executing them. In addition to characterizing the programs as a whole, we would like to separate the portions of the programs that have regular parallelism from the rest of the code and characterize the latter, for reasons discussed above. To carry out such a separation of code and characterization, an appropriate experimental system of code generation and execution is necessary. A vector machine provides an excellent setting for such characterization, since it separates out the regular parallelism in the code from the rest — program portions that have regular parallelism are executed via vector instructions, while the rest of the code is executed via normal scalar instructions. Furthermore, while all of the regular parallelism in the programs may not be thus separated and exploited, the actual separation achieved is a reflection of the current state of the art of compiler and processor technology.

In order to use a current vector machine as an experimental setup for characterizing scientific codes in general, it is necessary to show that vector machines are currently a good paradigm for executing scientific code. A very active area of research currently is the instruction-issue stage parallelism in uniprocessors. As mentioned earlier, there is no consensus yet on the amount of such parallelism necessary. Given this situation, it is necessary to address this topic with regard to any particular machine chosen for a study that aims to be useful beyond the scope of that particular machine. In particular, for our study, if the single instruction issue limit in a current vector machine is a performance bottleneck, then a VLIW machine or a superscalar machine would provide a more appropriate setting for our studies. For example, if the large issue bandwidth of a VLIW machine is well-utilized for a given resource architecture throughout the execution of programs, then certainly vector machines with instruction issue limited to one instruction per cycle do not exploit the parallelism in programs well. Below we discuss in some detail the execution of programs by vector machines, in an attempt to address the above question.

The work of a vector instruction is equivalent to that of several scalar instructions: a vector instruction executes several *operations*, one for each element of its result vector. Thus, the peak *operation issue bandwidth* of a vector machine is much higher than one. Every vector instruction in progress issues an operation each clock cycle; thus, if n vector instructions are in progress simultaneously, we have an operation issue rate of n operations per clock cycle. Furthermore, vector functional units implicitly carry out array indexing and loop control operations, thus avoiding the additional issue bandwidth that would have been necessary if the same work were implemented by a non-vector code fragment. Thus, operation issue parallelism needed for executing data-parallel code is available in a vector machine. Empirical evidence is shown in [Tang88] which indicates that in the Cray machines the quantity of vector hardware resources, rather than the instruction issue stage, is currently the performance bottleneck (i.e., currently, additional instructions can not be issued on these machines due to the lack of free vector functional units and vector registers).

The difference between the current vector machines and the multiple instruction issue machines, with respect to peak instruction issue parallelism, lies in the number of *scalar* instructions that can be issued simultaneously in a clock cycle. The vector machines can only issue one scalar instruction per clock cycle, while the multiple-instruction-issue machines can issue multiple scalar instructions per clock cycle. Current vector machines, however, have comparatively deep pipelines even in the scalar functional units (and in the memory pipeline, which is 14 clocks on the CRAY X-MP, for example) as will be shown in the next chapter. As discussed earlier in this chapter, deeper pipelines imply less need for parallelism in the instruction issue stage. On the other hand, deeper pipelines imply slightly longer latencies for the functional units (due to the effects of pipelining [Kunkel86]), and shallower pipelines might be more desirable for relatively less-parallel code. We note that there has been no quantification of the availability of parallelism in the scalar code of vector machines.

In conclusion, vector machines execute data-parallel code very efficiently. Superscalar and VLIW machines can also exploit such parallelism, although they use a different model to specify such computation, and incur different costs in exploiting such parallelism. Since data-parallelism is the dominant form of parallelism in scientific code, vector machines are a very good choice for studying such predominantly data-parallel code. The execution of scalar code by current vector machines and current multiple-instruction issue machines is a subject of ongoing research. The scalar portions of current vector machines are not the best models for executing scalar code (later in this dissertation we address some of the issues involved here). However, choosing a vector machine provides us the opportunity of identifying and studying code that is not vectorizable by current state-of-the-art systems.

Having chosen the vector architecture as a sufficiently good paradigm for the execution of scientific code, we devote the rest of this thesis to understanding the characteristics of programs compiled for such machines. These studies provide a basis for designing the next generation of such processors, and could also provide insights into the design of other architectures.

Chapter 3

STUDY BACKGROUND AND METHODOLOGY

3.1. INTRODUCTION

This chapter provides the background for the studies reported in this thesis. First, in section 3.2, we give a very brief sketch of the studies we carry out in this dissertation. Here we discuss some issues involved in the studies of a vector machine; in particular, we discuss the desirability of a separate scalar unit in a vector machine and motivate the study of the scalar code in vectorized programs. Next, in section 3.3, since we study a specific machine, the CRAY Y-MP, we present an overview of the key features of the machine that are of interest from the point of view of our studies. In section 3.4, we discuss the benchmarks that we use in our study. In section 3.5, we discuss the measurement methods we use to collect data and our metrics, and a few caveats that have to be observed while using the data. We summarize the chapter in section 3.6.

3.2. STUDY AND DESIGN OF VECTOR MACHINES

In the previous chapter we concluded that vector machines provide a good architectural paradigm for scientific workloads. The architecture design process involves understanding the behavior of machines when running typical workloads, and we attempt to provide such an understanding of vector machines in the rest of this dissertation. We concentrate on studying vector architectures at the instruction level, focusing on an example machine, the CRAY Y-MP processor. The CRAY Y-MP is a state-of-the-art vector supercomputer equipped with a state-of-the-art vectorizing and optimizing compiler. We carry out our studies using the PERFECT Club programs as benchmarks. Both the processor and the benchmarks are discussed later in this chapter. No such study of a vector machine, using long-running applications as benchmarks, has been reported in the literature to date.

We study several issues that are relevant at the instruction-level — the instruction mix, nature of basic blocks, data-dependences, and several others which we do not list here. The importance of each of the several issues we study are discussed along with the presentation of the study of the issue. Since vector instructions execute several *operations* each, we distinguish between instructions and operations throughout our study. Operations and instructions are discussed in more detail in section 3.5 which discusses the study methodology.

The scalar and vector portions of programs have differing behaviors and resource-needs due to their different characteristics. Vector portions need high bandwidth due to the large amounts of data parallelism in them; long functional-unit and memory latencies can be tolerated by exploiting the large amounts of parallelism available in such code. Scalar portions, on the other hand, need short latencies since they have tight control- and data-dependencies; they have comparatively less parallelism, and hence find it difficult to tolerate long latencies via the technique of executing independent instructions during long latencies. Shorter latencies enable quicker resolution of dependencies and hence quicker instruction issue and program execution. Pipelining a functional unit tends to increase the overall latency of the unit, due to the additional overheads incurred such as the latching of results at the end of each pipeline stage[Kunkel86]. Therefore it would be desirable to have less pipelining in the scalar functional units.

Thus, a separate scalar unit with limited pipelining, which is tuned to the scalar code in the vectorized programs, seems desirable. Given this premise, we attempt to separate the scalar code in our benchmarks from the vector code and provide a characterization of the scalar code throughout our studies. We note that the Cray machines have separate scalar and vector portions, with different functional units for scalar and vector instructions and correspondingly different register sets. Thus it would be possible to tune each portion differently, to cater to respective codes. However, currently, floating-point functional units are shared by the vector and scalar instructions. This implies that these units have to be deeply pipelined for the benefit of the vector instructions. Furthermore, even the scalar functional units are pipelined to a fair extent, as will be shown in section 3.3, and this could hurt scalar performance, as discussed above.

After studying overall programs characteristics in Chapter 4 of this dissertation, we focus on several issues specifically relevant to scalar code in Chapter 5. We note that in addition to scalar and vector codes having different characteristics, the scalar code found in vector programs could have characteristics quite different from those of non-vectorized programs. This is due to the fact that in vectorized programs the data parallelism is eliminated, to an extent dependent on the level of vectorization achieved, from the scalar portions of the programs. We note that similar suggestions have been made in [Smith90]:

> "the basic character of scalar codes processed by a vector supercomputer may be quite different than in a computer without vectors, and this implies superscalar processing units that may also have a different character".

Several aspects of speeding up such scalar code in future vector supercomputers have been commented upon in the above paper. We address some issues that are relevant to such scalar code, in Chapter 5. Our experimental setting provides us with excellent examples of such scalar code. The code for the CRAY Y-MP is vectorized by a state-of-the-art compiler, and any scalar code in the programs is not currently automatically vectorizable. A study of such scalar code has not been reported to date.

3.3. OVERVIEW OF THE CRAY Y-MP PROCESSOR

We highlight some features of the CRAY Y-MP processor architecture [CRI88], 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 [CRI84a]; 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 CRAY-1[Russel78] and the CRAY-2[CRI85] are also similar to the CRAY Y-MP in many respects. The most significant differences among the different Cray machines are in the capacity, organization, and latency and bandwidth of the main memory system. The CRAY-2 also has a local memory in addition to main memory. The functional unit latencies vary to some extent among the machines, but functional unit architectures exhibit several similarities otherwise. The significant difference of the CRAY-2 is that it can be thought of as having double the clock rate for vector operations as compared to its scalar instructions. While several other vector supercomputers such as the NEC SX [Watana87], the HITACHI S-820 [Eoyang88], and the FUJITSU VP [Miura83] have data caches, none of the Cray machines have a data cache. Also, the NEC SX-2 [Watana87] and the NEC SX-X[HNSX89] have multiple *pipes* for each vector functional unit, i.e., there are for example 4 copies of each vector functional unit and when a vector instruction is issued each of the pipes operates on a quarter of the operations in parallel, thus reducing pipeline startup latency. However, on the whole, the vector supercomputers exhibit several similarities such as separate vector and scalar units, multiple vector and scalar functional units, multiple vector registers, etc. Thus, our study of the CRAY Y-MP sheds light on several issues common to many vector supercomputers.

We now discuss some features of the CRAY Y-MP that are of interest to our study. Figure 3.1 [CRI84] shows some of the key features of the processor. The processor is 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 comprising of its own set of functional units. The vector unit, address unit, and scalar-computation unit each have an individual primary register set — a set of eight vector (V) registers with 64 elements each, a set of eight 32-bit scalar (S) registers, and a set of eight 24-bit address (A) registers, respectively. 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. We would like to list the specific latencies of the CRAY Y-MP, but this information is, as of date, considered Cray Research Inc. proprietary information. However, the CRAY Y-

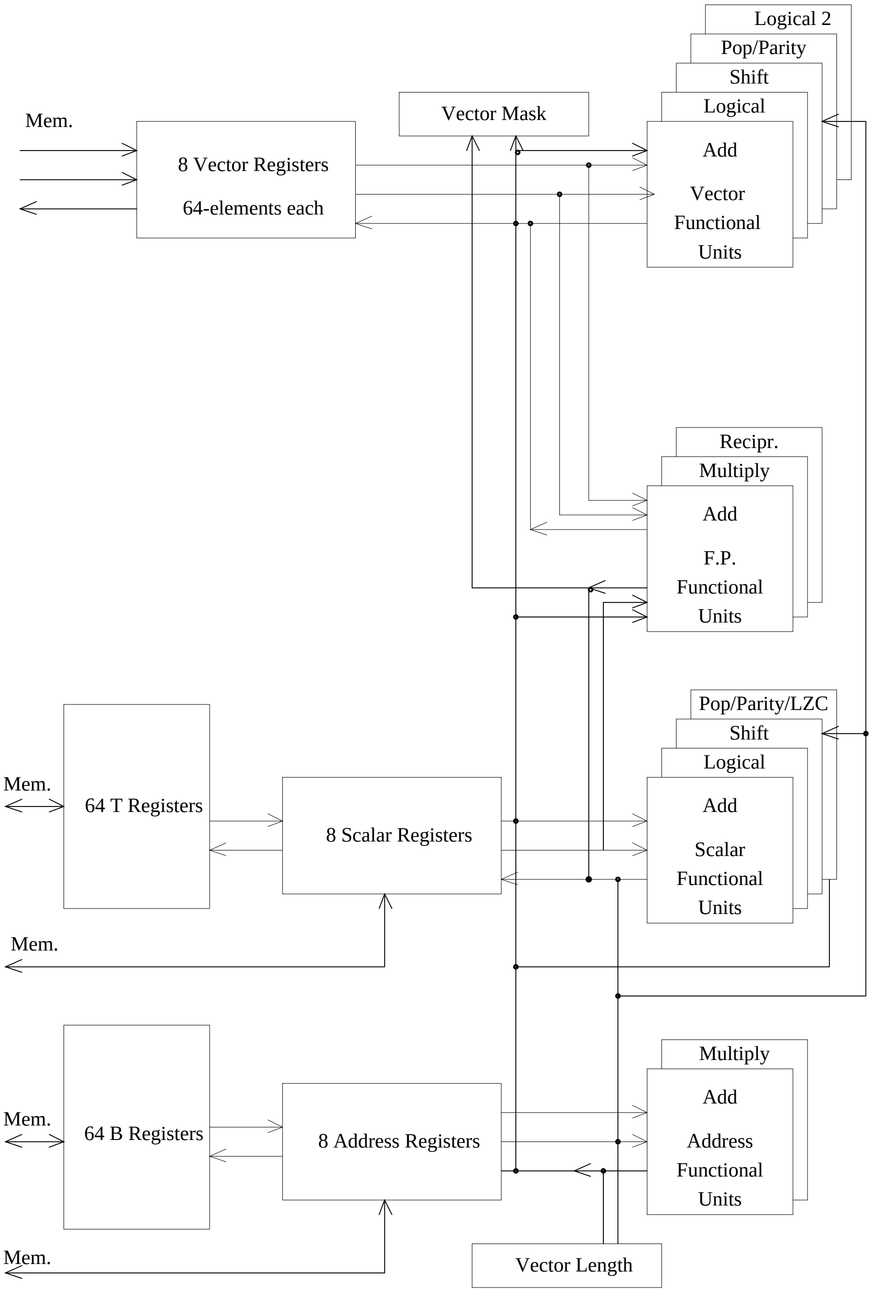


Figure 3.1: The Processor Architecture of the CRAY Y-MP.

(based on the CRAY Y-MP Reference Manual)

MP latencies are very similar to, but slightly longer than, the latencies of the CRAY X-MP. The specific functional unit latencies of the CRAY X-MP are listed in table 3.1. We note that the functional units have deep pipelines, and the memory latency is 14 clock cycles.

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. Data-transfer instructions are of three types: scalar transfers of single words between the primary registers and main memory, block-scalar transfers of a block of words between the scalar backup registers and main memory, and vector transfers of vectors between the vector registers and main memory. All computation instructions are register-register instructions that operate on primary registers. The secondary registers do not have direct data paths to the functional units; they have to be transferred to the primary registers before they

Inst. Type	Latency (clocks)
FP_ADD	6
FP_MUL	7
S_ADD	3
A_ADD	2
A_MUL	4
S_SHIFT	3
RECIPR	14
POP_LZC	4
LOAD	14

Table 3.1: Functional-Unit Latencies for the CRAY X-MP

can be used by a computation instruction. 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. The instruction cache is 512 parcels, or 1K byes, long, and is partitioned into four instruction buffers (cache blocks) of 128 parcels or 256 bytes each. The instruction cache has a dedicated port to the main memory.

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 [Smith83]. (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 this thesis.

3.4. BENCHMARKS

We use the PERFECT Club [Cybenk90] 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.

We choose these benchmarks as they are carefully-chosen long-running application programs rather than short kernels or loops. Table 3.2 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. We observe that each program executes hundreds of millions of CRAY Y-MP instructions, and takes hundreds of millions of clock cycles to run. DYFESM has the shortest running time at 647 million CRAY Y-MP clock cycles; FLO52 has the fewest number of instructions at 176 million. The largest execution time is 9 billion clock cycles (for MDG), and the largest instruction count is 3.4 billion instructions (for OCEAN). Overall, the benchmark set has 12.4 billion instructions and takes 49 billion clock cycles to execute. Thus, the benchmarks we study are long-running applications, as opposed to the kernels or small benchmarks that have been used so far for vector machine studies. For an instruction-level study such as ours, it is essential to study long-running, real programs in order to obtain a realistic picture of program behavior and machine utilization. While short kernels or loops are easy to handle, they do not completely represent program behavior. For example, the startup work prior to loop execution is not represented in such benchmarks. Furthermore, the widely-used Livermore FORTRAN Kernels are representative of only one workload (i.e., the workload at LLNL), while the PERFECT Club programs are drawn from a wide variety of of application workloads.

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
Total	49,075	12,453

Table 3.2: Benchmark Sizes — *Compiler-Optimized Version*

(as reported by the Hardware Performance Monitor)

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.

Although the Cray compiler is a state-of-the-art vectorizing and optimizing compiler, current state-of-the-art compiler technology for supercomputers is still unable to harness a significant portion of the power of the machines in many cases. It is very common for supercomputer users to hand-tune programs to their machines to harness more of the machine's power. Such hand-tuning includes reorganizing code to simplify the compiler's job, and inserting explicit compiler directives in the code to aid the compiler in its job. For example, consider a set of nested loops. Interchanging the loops could provide a vectorizable innermost loop where the original program had a non-vectorizable innermost loop. If the compiler cannot easily handle loop reordering, manually reorganizing the loops could provide tremendous performance improvements. As another example, if a loop contains ambiguous array references, usually the compiler has to be conservative and will not vectorize the loop. However, if the programmer knows that array references will never conflict, inserting a compiler directive that asserts that the loop is vectorizable enables the compiler to attempt vectorization. Since such hand-tuning of code is common for supercomputer applications, it would be interesting to compare and contrast code that is just compiler-optimized to code that is hand-tuned first and then automatically compiled.

We carry out precisely such a comparison in this thesis. We study two versions of the PERFECT Club benchmarks: one that is directly compiled by CFT77, and another that is hand-tuned and then compiled by CFT77. For the hand-tuned version of the PERFECT Club, we use the version, hand-tuned by a Cray Research, Inc. team of programmers, that won the 1990 "Gordon Bell-PERFECT" Award for the fastest supercomputer application. Thus, we study some of the best hand-tuned version of the PERFECT Club programs available to us.

Table 3.3 provides the execution times and the instruction counts for the hand-tuned version of the PERFECT Club programs. The execution times and the instruction counts of several programs show considerable improvement over their non-hand-optimized versions. The last column of table 3.3 shows the speedup obtained by hand-optimization. We note that DYFESM and FLO52 are excluded from the comparison of execution speeds. This is due to the fact that the hand-optimized versions of DYFESM and FLO52 available to us from Cray Research, Inc. are run on larger data sets than the compiler-optimized versions (the larger execution times we see for these two programs are due to the larger data sets). Ideally, one would like to use the exact same data-sets for both versions of the benchmarks. We tried to obtain runs of the programs on the same data-sets as the original versions, without immediate success; in the interests of time, we use the larger data-set versions. (We note that we do not have access to Cray Research, Inc.'s machines, and have to rely on employees of Cray Research for obtaining benchmarks or running programs.)

We note that the change in the two data-sets does not detract from our studies, since the issues of interest to us are the *proportions* of various events in each of the programs, rather than the absolute values or counts of the events. When we compute averages across programs, we use *normalized* values in order to provide equal importance to each program irrespective of its (execution) length, and hence the averages are also not affected. The PERFECT Club benchmarks are chosen from a variety of applications spread across several application centers and laboratories, rather than from a single workload (as for example the Livermore Loops which is drawn only from the application programs at the Lawrence Livermore Laboratories). The PERFECT Club report[Berry89] mentions such diversity to be one of its prime goals. Thus, the relative importance of the individual programs is not clear, and hence we choose to assign equal importance to all the programs.

The chief effect of data-set size on a vectorizable program is the following. As data-sets increase in size, the level of vectorization of the program increases, and hence program execution efficiency increases. The increase in data-size of the two programs in our benchmark suite does not raise much concern on this front: FLO52 and DYFESM show very nearly the same level of vectorization both before and after hand-optimization, as will be shown in the next chapter.

The speedups shown by the various programs reflect the efficacy of hand-optimizations and the amount of machine-power left unharnessed even by a state-of-the-art compiler. The set of programs as a whole shows executes 1.5 times faster, while a couple of individual programs execute 4 times as fast as the compiler-optimized versions.

In this dissertation we study the user routines of the benchmark programs. We do not include in our study the library routines executed by the benchmark programs. An important practical reason for this is the fact that the library routines are considered proprietary by Cray Research, Inc., and hence they are not available to us for study purposes. Furthermore, the early version, used by us, of a Cray Research, Inc. software tool necessary to study the programs did not provide information about library routines (the software tool is discussed in section 3.5). Beyond these practical factors, however, we consider the separation of library- and user-routines, in studies such as ours, important from a technical viewpoint, for the following reason. Many of

Benchmark	Time (millions of CPs)	Instructions (millions)	Speedup
ADM	1,245	421	3.60
ARC3D	5,712	1,167	1.01
BDNA	1,017	119	1.52
DYFESM	882	325	(NA)
FLO52	827	227	(NA)
MDG	6,066	933	1.49
MG3D	8,727	964	1.00
OCEAN	2,041	351	3.92
QCD	1,067	239	2.15
SPEC77	2,923	542	1.07
SPICE	390	96	4.06
TRACK	682	245	2.52
TRFD	779	343	1.67
Total	32,358	5,972	1.55*

(* : Speedup of the Benchmark Set with DYFESM and FLO52 excluded)

Table 3.3: Benchmark Sizes — *Hand-Optimized Version*

(as reported by the Hardware Performance Monitor)

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. Thus, by not including the library routines in the study, we focus on the performance of compiled FORTRAN code. Even if library routines are studied, it would be essential to present them separately from the user routines. When studying hand-optimized code, however, this distinction between user- and library-routines blurs to some extent. However, note that even the hand-optimized user-routines are written in high-level language, while the library routines are coded in assembly language which presumably provides more leverage over the machine.

3.4.1. Scalar Code in the Benchmarks

In section 3.2 we discussed the motivation for the design of a separate scalar processing unit geared to execute the scalar code found in vectorized programs. Briefly, the scalar code in vector programs limits the improvement attainable on the programs (Amdahl's Law), and hence it is important to speed up scalar code. Since the scalar code left over after vectorization could exhibit behavior very different from both vector code and the scalar code found in general-purpose (i.e., non-engineering programs), as discussed in section 3.2, it is important to characterize such code and base the design of the scalar unit in a vector machine on such characterization.

We attempt to provide a characterization of scalar code in this thesis. Here we address the issue of isolating scalar code from our benchmarks for the purpose of characterization.

One could define all the scalar instructions in the programs to comprise scalar code. However, scalar instructions whose execution is overlapped with the execution of vector instructions are essentially executed for free. One method of isolating "real" scalar code would involve simulating program execution and isolating all the scalar instructions that are in execution whenever all the vector units of the machine are idle. Since our study methodology avoids complete program simulation given the large programs we use as benchmarks, we use a different method for isolating scalar code. Furthermore, scalar code chosen in such a manner is to some extent dependent on the *particular* machine implementation being considered, while we are interested in a more general choice of scalar code. For example, changing the functional unit latencies or the number of vector pipes would change the amount of overlap between vector instructions and some scalar instructions.

Instead, we use the following method to isolate scalar code. We first observe that scientific programs are dominated by loops. In our benchmark programs, for example, the dynamic count of basic blocks is 3 to 4 orders of magnitude larger than the static count. This suggests the presence of loops with large loop counts. Tables 3.4 and 3.5 provide the static and dynamic basic block counts in the compiler-optimized and hand-optimized versions of our benchmarks. While the static block counts are in the few hundreds to tens of hundreds, the dynamic block counts are in the tens of millions. For example, the compiler-optimized version of mdg has 655 static basic blocks but the blocks are executed 84 million times. Thus, on the average, each loop is executed a few thousand times. Such execution could be due either to a high iteration count or to the nesting of the loop within other loops.

Furthermore, practically all of the vector computation instructions carry out the core of the program's computation and are executed inside loops. Such vector loops

Program	Static Blocks	Dynamic Blocks
ADM	1,325	52,329,814
ARC3D	1,199	48,984,993
BDNA	1,140	1,811,966
DYFESM	1,443	10,753,991
FLO52	1,234	8,219,900
MDG	655	84,305,047
MG3D	506	24,709,822
OCEAN	1,264	61,526,097
QCD	1,152	33,220,407
SPEC77	1,538	9,787,340
SPICE	1,699	29,117,818
TRACK	809	43,277,021
TRFD	316	40,845,277

Table 3.4: Static and Dynamic Basic Blocks Executed

— *Compiler-Optimized Codes*

Program	Static Blocks	Dynamic Blocks
ADM	1,736	10,618,381
ARC3D	1,225	55,954,584
BDNA	1,304	1,210,429
DYFESM	1,371	15,303,928
FLO52	1,298	11,818,539
MDG	542	29,049,143
MG3D	455	23,811,309
OCEAN	1,149	10,710,535
QCD	1,290	20,199,363
SPEC77	1,564	10,604,990
SPICE	2,814	5,640,153
TRACK	714	8,937,182
TRFD	258	27,810,474

Table 3.5: Static and Dynamic Basic Blocks Executed

— *Hand-Optimized Codes*

also contain some scalar instructions. It is very likely, however, that the execution of these scalar instructions is overlapped with that of vector instructions (either of the same loop iteration or of the previous iteration). For example, a vector addition of two arrays of 1000 elements each has to be carried out using a vector addition within a loop on the CRAY Y-MP since each vector instruction is limited to carry out a maximum of 64 operations (the maximum vector length). (We note that there could be a few cases where there exists tight dependencies across loop iterations such that the scalar instructions of the vector blocks are executed only when no vector instruction is in execution. Such instructions should ideally be considered as scalar code; our method of studying just the scalar basic blocks ignores these instructions.)

Other than the loops containing vector instructions, vectorized programs consist of scalar basic blocks that either set up work for the vector loops, or complete the work of a vector loop. In addition, the non-vectorized computations of the programs of course are found in the scalar blocks. All these scalar blocks are usually executed when no vector instruction is in progress (except that some times the vector instructions in the last loop iteration could overlap the execution of scalar code beyond the loop to some extent). Thus, we choose scalar code to be all the scalar basic blocks (i.e., basic blocks that contain no vector instructions) in the program. We hence exclude vector instructions and scalar instructions found in vectorized basic blocks. In the rest of the thesis, whenever we refer to scalar code in the benchmarks, we refer to code thus selected.

As pointed out by Amdahl's law, even if the scalar basic blocks contribute only a small portion of the program's instruction count, they significantly limit the execution speedup achievable. In chapter 4 we identify the fraction of basic blocks of our programs that are scalar, and discuss the reasons for the fractions seen, before proceeding with the characterization of the scalar code. Since the proportions of scalar and vector basic blocks are better understood given a perspective of overall program vectorization, we delay this discussion till chapter 4. Suffice it to say that even a small proportion of scalar code significantly limits performance improvement.

3.5. MEASUREMENT METHODOLOGY

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. 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, [Mitche88]). We use a Cray Research production software tool, JUMPTRACE [Kohn89], 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.

The version of JUMPTRACE used in our studies could instrument only the user routines of the programs. Partly due to this reason, and partly due to other reasons that are discussed in detail in section 3.4, we study only the user routines of our programs. The limitation of JUMPTRACE is due to the fact that library routines are written in assembly language (due to performance considerations). JUMPTRACE relies on some instrumentation code that is incorporated during compilation from FORTRAN down to the assembly language, and hence can not instrument assembly code. A modified version of JUMPTRACE, available since around April 1991, has the capability of tracing library routines. However, library routines are considered proprietary by Cray Research, Inc., and are not available to us for study. Assuming library routines are available for study, it would still be desirable to separate user-routines and library-routines in such a study; we discuss the reasons for this in section 3.4.

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 a non-obtrusive manner. For example, HPM monitors program execution time, instruction issue stage utilization, and the number of floating-point, integer, and memory operations executed.

3.5.1. Instructions and Operations

The instruction set of a vector processor consists of *scalar* and *vector* instructions. The scalar instructions of the CRAY Y-MP are similar to the instructions of any simple LOAD/STORE architecture. Vector instructions, on the other hand, initiate work equivalent to that initiated by several scalar instructions. For example, a vector load instruction might initiate 64 memory load operations, a vector add might initiate 64 floating-point operations. A scalar instruction initiates only one operation, except may be for an implicit indexing operation in memory instructions. Thus, when comparing vector machines with non-vector machines, it is desirable to compare operation counts rather than instruction counts. Furthermore, in many cases operation counts better reflect the utilization of machine resources. Hence, we distinguish between and provide both instruction and operation measurements in our studies.

To obtain operation counts, we expand each vector instruction into VL operations, where VL is the *vector length* of the vector instruction. VL specifies to the vector instruction the number of individual elements of the vector operands that are to be operated upon, and hence the number of operations to be executed. We note that scalar code that carries out the equivalent work of a vector instruction can require 2xVL or 3xVL operations, since it has to implement loop-control and array indexing in software (whereas these are implicitly handled by the hardware for a vector instruction). For example, consider the following vector instruction:

$$V1 = V2 + V3.$$

An equivalent scalar loop (in pseudo-assembly language) is:

```
        loopcntr = 64;
LOOP    V1[loopcntr] = V2[loopcntr] + V3[loopcntr];
        loopcntr = loopcntr - 1;  /* this instruction sets a condition code */
        BrG LOOP /* branch to LOOP if (loopcntr > 0) */
```

Thus, for each operation in the vector instruction above, we have 2 additional operations in the scalar loop. But in our studies we assume no additional operations for each vector operation; thus each vector instruction translates into VL operations.

3.5.2. 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. The number of operations executed by a vector instruction is determined at runtime on the CRAY Y-MP, 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. Furthermore, since we do not simulate program execution in entirety, our study methodology does not provide an opportunity for the use of such information even if it were available. Instead, we use the average vector lengths reported by the HPM for each of the benchmark programs. HPM reports the average vector length over the execution of a program for three different vector instruction classes — floating-point, integer, and memory instruction classes. Tables 3.6 and 3.7 present these measurements of the vector lengths for the compiler-optimized and the hand-optimized programs. We see that the average vector length is usually much less than 64 — the maximum vector length supported by the CRAY Y-MP; in extreme cases the average is as low as just a few elements (for example, in the case of SPICE, ADM, and TRACK). The average vector lengths depend on the the size of the data set to be operated upon. Often the number of elements to be operated on at a time can be much less than the hardware-supported maximum of 64. Furthermore, short vector lengths can be generated in another context. Whenever an array to be operated on is larger than 64 elements in size, the array is strip-mined into 64-element chunks and into a smaller chunk that accounts for any left over elements. This final chunk thus has a vector length shorter than 64.

There exists a small margin for error in our methodology due to the 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

Program	Average Vector Length		
	Integer Inst.s	F.P. Inst.s	Memory Inst.s
ADM	25.44	7.02	11.52
ARC3D	38.33	32.12	31.81
BDNA	39.81	50.99	47.62
DYFESM	34.31	21.19	14.53
FLO52	43.62	40.52	40.89
MDG	37.16	41.53	45.44
MG3D	60.57	61.77	62.27
OCEAN	47	41.58	48.84
QCD	12.17	13.15	13.15
SPEC77	34.46	27.34	29.98
SPICE	13.05	2.01	17.03
TRACK	5.75	5.32	26.01
TRFD	18.44	22.13	22.20

Table 3.6: Average Vector Length as reported by HPM

— *Compiler-Optimized Codes*

Program	Average Vector Length		
	Integer Inst.s	F.P. Inst.s	Memory Inst.s
ADM	16.42	18.56	18.73
ARC3D	38.40	31.78	31.39
BDNA	44.39	50.55	47.75
DYFESM	9.72	19.27	15.35
FLO52	41.73	39.69	40.58
MDG	36.55	48.38	47.76
MG3D	50.12	61.73	62.42
OCEAN	47.81	40.23	45.22
QCD	28.09	22.94	25.26
SPEC77	29.72	29.36	31.48
SPICE	42.92	41.15	43.95
TRACK	36.36	11.99	21.48
TRFD	21.39	22.27	20.99

Table 3.7: Average Vector Length as reported by HPM
— *Hand-Optimized Codes*

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.

We note that if the distribution of vector lengths is highly skewed then using the average vector length could result in comparatively larger errors. Such a situation is precluded to a large extent due to the following reasons. The compiler incorporates a rule that prevents the generation of vector instructions when very short vector lengths are involved, since vector instructions are inefficient at very short vector lengths.

Instead, the compiler uses scalar loops coupled with loop unrolling to achieve good performance. Regarding strip-mining, a vector length shorter than 64 is generated only once for each strip-mined array, whereas a vector length of 64 is generated from one to many times for the array depending on its size relative to 64. Therefore the average vector length in this situation will more likely be closer to 64 on the average. Therefore the vector lengths can be expected to be not highly skewed in the programs, and thus the average can be expected to be a fairly accurate measure of the actual vector lengths. We note that a few of the average vector lengths reported are smaller than 10 instructions. This is due to the fact that when the vector length is not known at compiler time the compiler generates code to dynamically determine the vector length. In these situations, although the average vector length is small, the individual vector lengths will be clustered close to that average, due to the heuristics used by the compiler to generate vector instructions with dynamically computed vector lengths. Furthermore, the compiler optimized versions of programs such as SPICE and TRACK are essentially scalar programs that have few vector instructions in them, as will be shown in the next chapter. Thus most of the very short vector lengths reported also have very little impact on the data collected.

We note in passing that simulating the programs in their entirety — the alternative to using average vector lengths — 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 programs themselves have long execution times (Table 3.2). 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.

3.6. SUMMARY

We presented in this chapter a discussion of some aspects involved in the study of a vector machine, to provide background to the study presented in this dissertation. Then, we presented an overview of the specific machine we study, a single processor of the CRAY Y-MP. We discussed the benchmarks we use, and the two optimization levels of the benchmarks that we consider. Finally, we described our study methodology and metrics, and discussed some caveats that have to be observed while using the data presented.

The next two chapters present the various studies of the CRAY Y-MP carried out using the methodology described here.

Chapter 4

CHARACTERIZATION OF VECTOR MACHINE PROGRAMS

4.1. INTRODUCTION

In this chapter we characterize the instruction-level behavior of programs executing on a single processor of the CRAY Y-MP. Our choice of the PERFECT Club programs as benchmarks for our studies was discussed in chapter 3. Throughout the instruction-level characterization, we compare compiler optimized programs with hand-optimized programs, in order to study the effects of hand-optimization on program characteristics. Furthermore, we also separate out the scalar basic blocks of the programs and study their characteristics in isolation. The motivation for such a study is the desirability of a scalar processing unit exclusively executing scalar blocks, as was discussed in chapter 3.

The methodology used in our study is discussed in chapter 3. This chapter is organized as follows. In section 4.2 of this chapter, we discuss the usage of various instructions and operations by the programs, and the level of vectorization of the benchmarks. In section 4.3, we discuss the sizes and the vectorization of basic blocks of the benchmarks. In section 4.4, we look at the number of user-routines and library-routines invoked by the two versions of the benchmarks. In section 4.5, we investigate instruction and operation issue rates. The significance of each of these issues is discussed in the sections that report the corresponding studies. In section 4.6, we summarize the studies reported in this chapter.

4.2. INSTRUCTION AND OPERATION MIX STUDIES

We present in this section several measurements pertaining to program vectorization, the CRAY Y-MP processor's instruction set usage, and operation execution counts. In the following subsections we point out the importance of these issues, present and analyze data, and discuss their implications on machine design.

4.2.1. Program Vectorization

The vectorization of a program is one of the key determinants of performance on a vector machine. Vector instructions, targeted at the data-parallel sections of a program, each execute on a series of data elements in a pipelined fashion. They completely hide functional-unit latency with pipeline parallelism, except for the pipeline startup overhead for the first operation of the instruction. Vector instructions perform work more efficiently than a corresponding scalar loop, as discussed in chapter 3 (also see, for example, Chapter 7 in [Hennes]). Therefore, 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* in the program, although, as per our experience, the relationship may not be linear. The fraction of program execution time spent executing vector instructions 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 dissertation (as discussed in detail in chapter 3), we will rely on the dynamic frequency of operations executed by 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 [McMaho86]). However, scientific programs could be inherently unvectorizable due to the presence of recurrences (data-dependencies across the iterations of a loop) in the code, ambiguous array subscripts that prevent the resolution of data dependencies at compile-time, data-dependent branches 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 it were compiled by hand. Hence, we first measure 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. Then, we examine the vectorization of the same programs after they are hand-optimized to improve performance. Hand-optimization includes rewriting of portions of the programs to ease their vectorization by the compiler, and insertion of specific compiler directives to vectorize portions that a compiler would otherwise not recognize as vectorizable. For example, a loop containing accesses to arrays via subscripts that are ambiguous to the compiler will not be vectorized unless a compiler directive asserts that the loop is vectorizable. Several techniques for improving the vetorization of and parallelism in programs are discussed in the literature, for example in [Padua86, Allen87].

As mentioned in Chapter 3, we study a version of the PERFECT Club benchmarks that is hand-optimized by a group of Cray Research programmers. Thus, since we are not privy to the optimizations carried out, we study only the behavior of the hand-optimized codes and do not attribute any behavioral changes to specific optimizations carried out. Throughout this chapter we study changes in program behavior due to hand optimization. In particular, in this section we compare the vectorization level of our benchmarks before and after hand-optimization; this study reflects the gap between performance delivered by a current state-of-the-art vectorizing compiler and performance achievable by excellent manual optimizations.

Table 4.1 presents the vectorization level of the automatically compiled programs. The last column of table 4.1 presents the fraction of all operations of each program vectorized in the compiler optimized version of the programs. We observe that

Bench-mark	Operation Class						
	Floating-Point		Integer		Memory		All
	% of F.P. Ops.	% of All Ops.	% of Int. Ops.	% of All Ops.	% of Mem. 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 4.1: Percentage Vectorization of Various Operation Classes

— *Compiler-Optimized Codes*

Bench-	Operation Class							
	Floating-Point		Integer		Memory		All	
mark	% of F.P. Ops.	% of All Ops.	% of Int. Ops.	% of All Ops.	% of Mem. Ops.	% of All Ops.	%	addnl. vect.
BDNA	99.8	49.9	97.4	3.9	99.8	43.4	97.2	+1.1%
MG3D	100.0	32.9	45.0	0.4	99.9	61.2	94.5	-0.6%
MDG	99.7	35.5	95.9	17.8	99.7	41.0	94.2	+6.5%
ARC3D	99.9	46.6	25.7	0.3	99.8	45.2	92.0	+0.9%
OCEAN	99.3	29.2	78.6	3.2	99.3	58.9	91.2	+48.4%
SPEC77	98.5	37.3	74.3	1.6	99.2	51.4	90.4	+0.1%
FLO52	98.3	38.8	50.8	1.2	99.4	48.8	88.7	-2.8%
SPICE	96.8	15.8	87.4	20.8	94.2	43.3	79.9	+68.4%
QCD	96.4	25.9	27.1	1.5	95.8	47.7	75.1	+70.9%
TRFD	99.9	32.7	17.0	0.5	97.3	40.5	73.7	+3.9%
DYFESM	95.9	29.4	41.2	3.5	92.8	32.7	65.6	-3.2%
ADM	89.2	29.0	32.6	2.6	85.3	28.0	59.6	+16.7%
TRACK	59.2	13.7	86.7	16.8	78.5	24.0	54.6	+40.2%
A.M.	94.8%		58.4%		95.5%		81.3%	

Table 4.2: Percentage Vectorization of Various Operation Classes
— *Hand-Optimized Codes*

the vectorization level is not around the commonly-expected 90% level, but spans the entire range, from about 96% all the way down to just 4% vectorization. BDNA is 96% vectorized, while only 4% of QCD's operations are executed by vector instructions. The limited vectorization of some of the programs could be due either to dependencies in the program or to the limitations of current state-of-the-art vectorization techniques. To examine this, we measure the vectorization level of the same programs when the compiler is supplemented with manual optimizations and rewriting/reorganization/hints for vectorization. The prior-to-last column of table 4.2 presents the fraction of all operations of each program vectorized in the hand-compiled programs. We see that the lowest vectorization level has jumped from 4% in the compiler optimized version to 54% in the hand optimized version. The programs OCEAN, TRACK, SPICE, and QCD exhibit tremendous increases in vectorization levels (as shown in the last column of table 4.2) — OCEAN improved from 43% vectorization to 91%, TRACK from 14% to 55%, SPICE from 11% to 80%, and QCD from 4% to 75%. While most other programs exhibit the same level of vectorization in the two versions, a few show a small drop. We note that this is quite possible when manual optimizations reduce the number of instructions executed by a program, for example by using more efficient code constructs. An intermediate result vector might be assigned to a register during hand-optimization, thus eliminating a vector load and a vector store instruction that was required in the compiler optimized version, and hence reducing the proportion of vector operations in the program. Also, some of the vector work of the program might be replaced by calls to more efficient vector library routines upon hand optimization; since we study only the user routines, we will see a corresponding decrease in vectorization level of the user routines. Thus, although the vectorization level is slightly lower for some programs, program execution is much faster due to the reasons just mentioned.

An important conclusion from the change in vectorization level seen is that performance delivered by state-of-the-art compilers can be quite short of performance achievable by manual optimizations. For some applications, the compiler may be unable to even partially vectorize programs that are actually vectorizable to a large extent (as, for example, in the case of SPICE and QCD). Knowledge on the part of the programmer of the underlying vector architecture and implementation is thus still essential to obtain the maximum performance from the machine, at least for some of the applications. We note that it is quite common today for supercomputer users to hand-tune applications to obtain better performance from the machine.

We have used the fraction of all program operations vectorized as our vectorization metric above, 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 [Hennes]. To investigate the representativeness of this metric, we examine the vectorization of various classes of operations in the compiler optimized version of the benchmarks (see table 4.1 again). Note again that the fractions presented in table 4.1 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 in the compiler optimized version. (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. However, for the hand-optimized programs, floating-point vectorization is much higher than overall vectorization. Except for TRACK, all the hand-optimized programs have at least 90% of their floating-point operations vectorized. Thus, floating-point vectorization is not a good metric of program performance. Furthermore, as seen from tables 4.1 and 4.2 again, the fraction of neither integer nor memory operations vectorized is very correlated with the fraction of all operations vectorized for either benchmark version, 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. Thus, the vectorization levels of individual classes of operations are not representative of overall program vectorization.

We observe that the fraction of memory operations vectorized in the compiler optimized version is quite high for ten of the thirteen original programs, with the lowest level for them being 79.3%. Of the other three programs, TRACK has 48.6% of its memory operations vectorized while SPICE has 32.6% of its memory operations vectorized; QCD has only 16.5% of its memory operations vectorized. For the hand-optimized programs, the fraction of memory operations vectorized is high for all programs; TRACK has the lowest vectorization of memory operations, at 78.5%. This clearly emphasizes the need for large memory bandwidth in vector supercomputers, even for programs that are not hand-optimized. Memory latency is relatively less important for the vector memory operations — the latency is hidden by the parallelism available, by pipelining memory accesses. One reason memory references of even scalar programs are vectorized to a significant extent might be that vector loads can be used to feed some of the data for scalar computations such as linear recurrences, as shown in [Sohi90]. We note that the CRAY Y-MP has 3 memory ports dedicated to data transfers, multiple memory banks, and pipelined memory access, to provide large memory bandwidth.

The reason for the high level of vectorization of memory operations is of course the high predictability of memory references. Such predictability can be exploited even in non-vector machines via intelligent data-prefetching (either to a data-cache or to registers) provided the memory bandwidth is available. When scalar programs have predictable memory references, a programmable cache[Sohi90] that exploits the knowledge of the memory access patterns can be used to reduce memory latency. Thus, the data suggest that scientific applications in general can utilize large memory bandwidth, even when compilers don't vectorize the overall programs to the maximum extent possible. Long memory latency is consequently less of a concern when it can thus be tolerated by using the parallelism in the predictable memory references or reduced by exploiting the predictability of the memory references.

Let us consider the vectorization of integer operations. We observe that the vectorized integer operations constitute only a small portion of all program operations.

The vectorization of the integer operations is spread across a wide range, and shows no correlation with the vectorization of either floating-point operations or the entire program. (As mentioned in Chapter 3, integer operations do not include address computations since a separate scalar address computation functional-unit is provided on the Cray machines and they are hence classified separately.)

Overall, we can conclude that the *average* fraction of operations vectorized for both hand-optimized and just compiler-optimized scientific programs is much less than the usually assumed 90%, insofar as the benchmarks are representative of scientific workloads. For our benchmark set, the average vectorization is 62% for the compiler-optimized programs, and 81% for the hand-optimized programs. (As discussed in Chapter 3, we assume all programs are equally important when computing the averages. Hence we use the arithmetic mean here.)

Considering the clustering of numbers in the overall vectorization columns of tables 4.1 and 4.2, we partition the benchmark programs into various classes in the rest of this dissertation: *highly-vector*, *moderately-vector*, and *scalar* programs for the original benchmarks, and *highly-vector* and *moderately-vector* classes for the hand-optimized benchmarks. We will present data only for these three classes of programs in the rest of this dissertation; space constraints and the volume of data prevent us from presenting information for the individual programs. We believe that classifying the programs as above minimizes the loss of information caused by considering only averages of numbers, since each group contains programs with very similar characteristics. Additionally, the classification helps us identify certain characteristics of the individual classes. We classify the programs as follows. For the original benchmarks, 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 classes; we use information about instruction-issue stall times, presented in section 4.5, to push them into the moderately-vector category.) For the hand-optimized benchmarks, ARC3D, BDNA, FLO52, MDG, MG3D, OCEAN, and SPEC77 are the *vector* benchmarks; and ADM, DYFESM, QCD, SPICE, TRACK, and TRFD make up the *moderately-vector* benchmarks.

4.2.2. Instruction Usage and Operation Counts

In this subsection, we discuss the CRAY Y-MP *instruction set* usage by the benchmarks and the *operation* mix in them. We consider both the compiler-optimized and the hand-optimized versions of the benchmarks, as well as just the scalar basic blocks of both the versions. As discussed in Chapter 3, we determine a benchmark class's usage of an instruction by averaging the *normalized* usage of the instruction by each of the programs in the class; thus, all the programs of a class are given equal importance, irrespective of the number of instructions executed by them individually. The numbers for operation usage are computed by expanding each vector instruction into the number of operations that it executes, by using the average vector length reported by HPM for each program. (This was discussed in detail in chapter 3.) 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.

Let us first consider the original compiler optimized benchmarks. Table 4.3 classifies the operations executed by the original benchmarks into various broad operation classes that are present in several architectures. For example, from the table we see that 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 obtained for other machines.

From table 4.3, we observe that overall the vector benchmarks consist almost entirely of floating-point operations and memory references. On the other hand, the scalar benchmarks have comparable amounts of floating-point and integer operations; and, for the moderately-vector programs, floating-point operations are three times as frequent as integer operations. We note that by integer operations we mean only those operations that are executed by the scalar-computation unit. Address-

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
Total	100%	100%	100%

Table 4.3: Percentage of Operations in each Operation Class

— *Compiler-Optimized Codes*

computation instructions, while also being integer operations, are executed by the address-computation unit and are classified separately. (Please see Chapter 3 for a discussion of the processor.) However, scalar integer operations are sometimes used to perform 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 (as we do not keep track of cross-block dependences), 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 on the Cray machines for maintaining loop counters. We observe that 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 memory 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%.

Miscellaneous operations, which are specific to the Cray machines, form a large fraction of the scalar and the moderately-vector benchmarks. We will discuss these operations in detail shortly. Branches, on the other hand, constitute a small fraction of the overall operation count, even for the scalar programs. We will discuss these operations also in detail shortly.

Let us now consider the effects of hand-optimization on the nature of the programs. Table 4.4 presents the operation distribution in the hand-optimized version of our benchmarks. Interestingly, although hand-optimization includes additional vectorization that reduces the overall operation count (as discussed in chapter 3), and the possible rewriting of code segments that might change the characteristics of the segments, we observe that the proportion of various operations within the individual benchmark classes remains very similar to that in original version. (The program operation counts and execution time have decreased with hand optimization, as discussed before in the previous chapter.) The moderately-vector benchmarks of the hand-optimized version have a slightly higher vectorization level than that of the original version. There are no other significant differences between the operation distributions in the two versions of each of the individual benchmark classes, except for the

Operation Class	Benchmark Subclass	
	Moderate	Vector
All FP	27.02	38.83
All Memory	39.38	50.19
All Integer	11.35	4.72
Address Comp.	5.20	1.85
Miscellaneous	15.24	4.05
Branches	1.77	0.31
Total	100%	100%

Table 4.4: Percentage of Operations in each Operation Class

— *Hand-Optimized Codes*

fact that more of the operations of each operation-type are executed by vector rather than scalar instructions. Thus, the additional hand-optimizations have not changed the overall proportion of various instructions and operations in the moderately-vector and vector classes of programs. Note, however, that the scalar class of programs has been eliminated by the hand optimizations. One significant implication of the above data is that since the instruction and operation mix is not affected by hand-optimization for a particular program class, processor design could be expected to be not much affected by whether the processor is targeted at compiler optimized or hand optimized programs *of a particular class of programs*. Note, however, that the hand-optimized programs have no programs of the scalar class. An important caveat however is that this conclusion is drawn only for the hand optimizations carried out in the programs we study here. As mentioned elsewhere, we are not privy to the optimizations carried out and hence are not able to discuss them.

Earlier in Chapter 3 we discussed the motivation for the design of a scalar processor unit tuned to the scalar portions (in particular, the scalar basic blocks) in scientific code. Towards aiding the design of such a unit, we examine the characteristics of such scalar code in our programs, and compare it to the characteristics of the overall programs and to that of non-vectorizable programs in general, throughout this chapter. We examine several other aspects of these scalar basic blocks in detail in chapter 5. Before examining the characteristics of the scalar basic blocks, let us examine their frequency in the programs, to put the scalar blocks in perspective with regard to overall program size. Tables 4.5 and 4.6 show the fraction of static and dynamic basic blocks of the programs that have no vector instructions in them. We observe that these dynamic scalar blocks are the majority for 10 of the 13 programs. Even among the vectorized programs, three of them (MG3D, BDNA, SPEC77) have approximately an equal proportion of dynamic scalar and vector blocks, while the other three (FLO52, MDG, ARC3D) have at least 70% dynamic scalar blocks. For the moderately-vector programs, the proportion of dynamic scalar blocks is at least 80%, and for the scalar programs, most of the blocks are of course scalar. Among the static basic blocks, the proportion of scalar blocks is even more for most of the programs. When the programs are hand-optimized the proportion of dynamic scalar blocks decreases

pgm	static (%)	dynamic (%)
MG3D	90.91	44.25
BDNA	85.26	53.74
SPEC77	81.99	57.95
FLO52	90.52	70.87
MDG	75.88	72.89
ARC3D	87.82	75.79
DYFESM	91.20	80.27
TRFD	95.57	82.58
ADM	91.62	87.97
OCEAN	92.96	90.20
SPICE	98.35	97.42
QCD	95.49	97.77
TRACK	95.18	98.81

Table 4.5: Proportion of Basic Blocks that are Scalar
— *Compiler-Optimized Codes*

by various amounts for individual programs, but still shows very similar overall characteristics for the two benchmark classes. (We remind the reader that the number of basic blocks changes, as shown in Chapter 3). The vector and moderately-vector programs still exhibit a high proportion of vectorized operations despite this proportion of scalar blocks because most of the dynamic scalar blocks are small in size while the dynamic vector blocks are large and contain many vector instructions. We will discuss the sizes and vector instruction content of basic blocks in greater detail in the next section (section 4.3).

Table 4.7 presents the operation mix in the *scalar basic blocks* of the original version of the benchmarks. If we consider the scalar blocks of all the thirteen programs together, then the floating-point, integer, memory, and address-computation operations are equally significant in number. In the scalar programs alone, however, the scalar basic blocks expectedly carry out the bulk of the program's work as there are

pgm	static (%)	dynamic (%)
MG3D	91.65	36.70
BDNA	85.81	44.59
MDG	74.54	49.07
OCEAN	92.08	59.09
SPEC77	81.33	60.31
FLO52	90.37	74.92
ARC3D	86.61	77.00
DYFESM	92.56	78.10
TRFD	94.57	80.96
TRACK	91.60	86.81
ADM	91.01	88.64
QCD	91.01	91.29
SPICE	83.87	91.75

Table 4.6: Proportion of Basic Blocks that are Scalar

— *Hand-Optimized Codes*

less than 3% vector blocks. Hence we see a higher proportion of both floating-point and memory operations in these blocks. We also observe that the scalar programs in entirety had a high proportion of memory operations at 25% (table 4.3), while the proportion is around 18% in the scalar blocks of these programs. This suggests that the few vector blocks in the scalar programs contain a significant number of vector memory instructions and operations.

The scalar blocks of the hand-optimized programs exhibit quite similar behavior on the whole, as seen from table 4.8. We note that the floating-point operations are a little less frequent when compared to compiler optimized codes. Most of the main work of the programs is now done by the vector basic blocks since the vectorization level of many of the programs has increased due to the hand-optimizations. Thus, for the scalar basic blocks of the programs, the proportion of floating-point units is smaller.

For the hand-optimized vector programs, branches are higher in proportion compared to the compiler optimized version, for very similar reasons. If the miscellaneous operations, which are specific to the Cray machines, are discounted, the branch frequency in the scalar basic blocks would increase to about 17%, comparable to the branch frequency in general-purpose (i.e, non-scientific) scalar codes.

The miscellaneous operations, which are due to the register file architecture of the Cray machines, are uniformly high in proportion in the scalar blocks for all classes of both compiler-optimized and hand-optimize programs.

Let us now consider the frequencies of specific operation types in the programs. These frequencies determine the importance of the different operations types and functional units and thus have affect processor design greatly. Table 4.9 subdivides the information presented in table 4.3 about operation frequencies in the compiler-optimized programs into classes that correspond to the functional units present in the CRAY Y-MP. It also presents instruction frequencies. Table 4.10 similarly presents data for the hand-optimized programs. Before examining the data further, we note that for BLK_LD and BLK_ST instructions, which 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. We note that this might affect the measurements of the proportion of the other operation types. However, we note that there are few BLK_LD/BLK_ST instructions; also, for many of the BLK_ST instructions the number of operations executed is one since these instructions are used as syncronization instructions, as will be explained later. Thus, we expect the effect of this approximation on the data measurements to be small.

Operation Class	Scalar (%)	Moderate (%)	Vector (%)	All (%)
FP	14.33%	9.83%	9.06%	10.51%
Memory	17.65%	10.67%	13.93%	13.78%
Integer	12.00%	18.25%	12.24%	14.03%
Address Comp.	8.28%	13.17%	15.02%	12.89%
Miscellaneous	40.80%	43.77%	43.72%	43.05%
Branches	6.96%	4.34%	6.02%	5.72%
Total	100%	100%	100%	100%

Table 4.7: Percentage of Operations in Scalar Basic Blocks
— *Compiler-Optimized Codes*

Operation Class	Moderate (%)	Vector (%)	All (%)
FP	7.29%	6.54%	6.89%
Memory	12.34%	8.97%	10.52%
Integer	13.61%	14.72%	14.22%
Address Comp.	14.54%	14.99%	14.78%
Miscellaneous	45.07%	45.59%	45.35%
Branches	7.14%	9.20%	8.25%
Total	100%	100%	100%

Table 4.8: Percentage of Operations in Scalar Basic Blocks
— *Hand-Optimized Codes*

Furthermore, tables 4.11, 4.12, 4.13, and tables 4.14 and 4.15 list the frequencies of individual operation types in the scalar basic blocks of the various classes of the compiler-optimized and hand-optimized programs. (Note that instruction counts and operation counts are identical for scalar basic blocks.) The last column of these tables lists the fraction of operations of a particular operation type that occur in scalar basic blocks. For example, from table 4.11, 96.77% of all the scalar memory loads of the scalar programs (compiler-optimized version) occur in scalar basic blocks. Thus, the remaining 3.23% of the scalar load instructions occur in the vector basic blocks of the programs. The tables also list, in the last row, the fraction of all program instructions and operations contributed by the scalar blocks. For example, from the last row of table 4.11, 85.5% of all operations of the scalar programs are scalar operations that occur in scalar basic blocks. We note here that vector basic blocks also contain a few scalar operations in addition to the vector operations. Thus, not all of the remaining 14.5% program operations are vector operations. We discuss the individual operation types below, and extensively refer to the above tables, several times without an explicit pointer to the tables for the sake of clarity.

Register Transfers

Most strikingly, more than one-third of all *operations* of the scalar class of the original benchmarks are operations used to transfer values between the various register sets in the processor (the miscellaneous category in tables 4.3 and 4.9). (Please refer to Chapter 3 for a discussion of the register sets of the processor.) The proportion of these register transfer operations decreases as we move to the vectorized programs. For the scalar basic blocks of all classes of both the original and the hand-optimized benchmarks, however, close to half the operations are of this category (tables 4.7 and 4.8).

There are several reasons for the high proportion of miscellaneous operations in the programs. The backup registers on the CRAY Y-MP are used to temporarily hold values whenever the primary register sets are full, resulting in several *register spill* instructions. Register spilling normally refers to moving a live temporary value present in a register to memory, in order to make room in the register set for another value that will be accessed before the value being spilled. In the Cray machines, register spilling usually only results in a movement of values between the primary (A or S) register set and the corresponding backup (B or T) register set, and we term these moves *spill instructions*. The pressure on the non-vector registers is of course higher in scalar code since there are more scalar computations in such code, and hence we see a higher proportion of spill instructions in scalar programs and in scalar basic blocks of moderately-vector and vector programs. Apart from register shortage, 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

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	0.255	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	0.254	0.32	1.29
Vector FP	V_FPADD	0.13	0.61	1.71	12.257	4.96	18.80
	V_FPMUL	0.17	0.59	1.259	10.47	5.16	19.44
	V_RECIP	0.03	0.06	0.00	0.05	0.26	1.04
Vector Mem	V_LD	0.251	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	0.251
	S_LOGIC	5.72	5.12	5.17	2.253	3.07	0.34
	S_SHIFT	0.63	0.59	1.255	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	1.251	0.17
	FP_ADD	7.17	6.56	5.22	2.92	1.88	0.20
	RECIPR	0.26	0.23	0.258	0.23	0.09	0.01
Mem	LD	10.250	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*	0.257	0.04*
Addr. Comp.	A_ADD	8.257	7.48	15.04	6.64	17.26	1.78
	A_MUL	0.16	0.14	0.91	0.255	0.41	0.04
Branches	BR	6.66	5.94	3.68	1.52	3.27	0.254
Misc.	MOV	12.14	11.00	14.12	5.88	21.85	2.23
	A_SPILL	7.251	6.64	9.66	4.64	9.94	1.08
	S_SPILL	21.75	19.93	17.49	8.51	8.82	0.94
TOTAL		100%	100%	100%	100%	100%	100%

Table 4.9: The Proportion of Instructions and Operations of Various Types — *Compiler-Optimized Benchmarks*

Operation Class	Instruction Class	Benchmark Class			
		Vector		Moderate	
		Insts.	Ops.	Insts.	Ops.
Vector Int.	V_LOGIC	0.85	2.53	0.50	4.98
	V_SHIFT	0.18	0.57	0.15	1.14
	V_INTAD	0.33	0.95	0.14	1.49
Vector FP	V_FPADD	5.41	19.27	1.76	12.50
	V_FPMUL	5.04	18.22	1.66	11.65
	V_RECIP	0.30	1.10	0.03	0.25
Vector Mem	V_LD	8.02	30.81	2.77	20.06
	V_ST	4.58	17.64	1.44	10.69
	V_GATH	0.32	1.17	0.53	4.35
	V_SCAT	0.10	0.36	0.10	0.96
Int	S_ADD	3.12	0.31	4.88	1.62
	S_LOGIC	3.24	0.32	5.41	1.79
	S_SHIFT	0.37	0.04	0.80	0.29
	POP_LZC	0.04	0.00	0.12	0.04
FP	FP_MUL	0.98	0.12	2.98	1.32
	FP_ADD	1.08	0.10	2.93	1.21
	RECIPR	0.14	0.02	0.24	0.09
Mem	LD	0.97	0.10	5.02	1.77
	ST	0.62	0.06	2.71	1.04
	BLK_LD	0.05	0.00	0.45	0.18
	BLK_ST	0.55	0.05	0.94	0.33
Addr. Comp.	A_ADD	18.39	1.80	15.09	4.95
	A_MUL	0.50	0.05	0.70	0.25
Branches	BR	3.10	0.31	5.35	1.77
Misc.	MOV	24.21	2.29	20.98	7.07
	A_SPILL	10.65	1.10	9.75	3.30
	S_SPILL	6.70	0.66	12.45	4.87
TOTAL		100%	100%	100%	100%

Table 4.10: The Proportion of Instructions and Operations of Various Types — *Hand-Optimized Benchmarks*

Inst.	Percent of			
Type	Scalar Insts.	All Insts.	All Ops.	This Inst.Type
LD	10.67	10.08	9.03	96.77
ST	5.39	5.09	4.59	96.84
BLK_LD	0.95	0.90	0.84	100.00
BLK_ST	0.64	0.61	0.56	97.22
FP_MUL	6.80	6.40	5.87	94.79
FP_ADD	7.27	6.85	6.27	95.74
RECIPR	0.26	0.25	0.22	97.47
S_ADD	5.32	5.01	4.61	98.03
S_SHIFT	0.65	0.61	0.57	95.34
S_LOGIC	6.01	5.65	5.06	98.53
POP_LZC	0.02	0.02	0.01	60.04
A_ADD	8.15	7.68	6.86	91.34
A_MUL	0.13	0.12	0.10	49.61
BR	6.96	6.54	5.83	97.93
MOV	11.53	10.91	9.91	89.38
A_SPILL	7.40	6.96	6.32	95.29
S_SPILL	21.87	20.57	18.85	95.73
Total	100%	94.27%	85.50%	

Table 4.11: Instruction Mix in the Scalar Basic Blocks

Scalar Programs — Compiler-Optimized Codes

Inst.	Percent of			
Type	Scalar Insts.	All Insts.	All Ops.	This Inst.Type
LD	6.73	5.07	2.51	98.84
ST	3.55	2.85	1.52	99.46
BLK_LD	0.23	0.17	0.09	100.00
BLK_ST	0.16	0.11	0.06	51.67
FP_MUL	3.58	2.94	1.72	97.73
FP_ADD	5.79	4.99	2.81	92.24
RECIPR	0.46	0.36	0.21	98.02
S_ADD	9.58	7.33	3.71	88.28
S_SHIFT	1.55	1.30	0.75	72.37
S_LOGIC	7.03	5.01	2.25	96.25
POP_LZC	0.09	0.08	0.05	86.98
A_ADD	12.51	8.98	4.19	63.34
A_MUL	0.66	0.44	0.17	45.55
BR	4.34	2.96	1.25	83.31
MOV	12.61	8.54	3.58	61.04
A_SPILL	9.50	6.70	3.15	71.92
S_SPILL	21.66	16.23	7.96	93.09
Total	100%	74.07%	35.98%	

Table 4.12: Instruction Mix in the Scalar Basic Blocks

Mod.-Vec. Pgms. — Compiler-Optimized Codes

Inst.	Percent of			
Type	Scalar Insts.	All Insts.	All Ops.	This Inst.Type
LD	9.82	3.24	0.36	92.28
ST	3.92	1.01	0.09	95.40
BLK_LD	0.12	0.03	0.00	100.00
BLK_ST	0.07	0.01	0.00	22.66
FP_MUL	3.95	1.29	0.17	98.71
FP_ADD	4.72	1.50	0.17	80.14
RECIPR	0.39	0.09	0.01	90.69
S_ADD	5.38	1.61	0.17	48.89
S_SHIFT	0.09	0.02	0.00	18.00
S_LOGIC	6.77	2.11	0.21	61.56
POP_LZC	0.00	0.00	0.00	21.91
A_ADD	14.79	4.89	0.56	31.38
A_MUL	0.23	0.07	0.00	38.57
BR	6.02	1.95	0.21	62.38
MOV	17.25	5.49	0.60	25.01
A_SPILL	8.61	2.83	0.32	28.98
S_SPILL	17.86	6.03	0.69	62.99
Total	100%	32.18%	3.55%	

Table 4.13: Instruction Mix in the Scalar Basic Blocks

Vector Programs — Compiler-Optimized Codes

Instruction	Percent of			
Type	Scalar Insts.	All Insts.	All Ops.	This Inst.Type
LD	7.23	4.69	1.62	94.25
ST	3.67	2.44	0.92	92.12
BLK_LD	0.68	0.45	0.18	100.00
BLK_ST	0.76	0.51	0.19	54.31
FP_MUL	3.77	2.62	1.15	91.75
FP_ADD	3.20	2.14	0.88	78.70
RECIPR	0.32	0.23	0.09	95.08
S_ADD	6.37	4.03	1.33	80.58
S_SHIFT	0.50	0.34	0.12	35.63
S_LOGIC	6.71	4.15	1.36	77.20
POP_LZC	0.03	0.03	0.00	39.36
A_ADD	14.09	8.61	2.72	54.45
A_MUL	0.45	0.24	0.08	31.07
BR	7.14	4.56	1.50	84.00
MOV	18.36	11.48	3.83	55.09
A_SPILL	11.18	7.07	2.36	72.16
S_SPILL	15.53	10.16	3.97	78.72
Total	99.99%	63.75%	22.3%	

Table 4.14: Instruction Mix in the Scalar Basic Blocks

Mod.-Vec. Pgms. — Hand-Optimized Codes

Instruction	Percent of			
Type	Scalar Insts.	All Insts.	All Ops.	This Inst.Type
LD	4.86	0.77	0.08	84.72
ST	3.73	0.55	0.06	87.24
BLK_LD	0.20	0.05	0.00	100.00
BLK_ST	0.18	0.04	0.00	6.51
FP_ADD	2.42	0.53	0.06	58.88
FP_MUL	3.60	0.96	0.12	96.05
RECIPR	0.52	0.14	0.02	92.94
S_ADD	5.31	1.23	0.13	36.20
S_SHIFT	0.22	0.06	0.00	18.36
S_LOGIC	9.19	2.25	0.24	65.87
POP_LZC	0.00	0.00	0.00	49.78
A_ADD	14.63	3.51	0.38	19.37
A_MUL	0.36	0.10	0.01	16.40
BR	9.20	1.63	0.17	55.48
MOV	19.14	4.61	0.50	19.26
A_SPILL	11.05	2.62	0.29	24.50
S_SPILL	15.40	3.98	0.45	57.43
Total	100.0%	23.0%	2.5%	

Table 4.15: Instruction Mix in the Scalar Basic Blocks

Vector Programs — Hand-Optimized Codes shuffle registers around so that the condition computed is finally stored in A0 or S0 before the branch is issued. We observe that about 27% of all operations in the scalar benchmarks of the automatically compiled programs are spill operations (20% are spills of the S registers, S_SPILLs, and 7% are spills of the A registers, A_SPILLs). 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 and can be maintained 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. Similarly, some of the scalar computations themselves have longer latencies than the address computations. 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.

We note that the hand-optimized codes have comparatively few spill instructions due to the fact that all the programs are at least moderately vectorized. We observe that the spill instructions form a smaller fraction of the miscellaneous operations (table 4.10), indicating possible improvement in register usage due to hand-optimizations.

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 and for memory cause long waits in the instruction-issue stage for dependencies to be resolved. Scalar code has a lot of control- and data-dependencies 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 if the spill instructions themselves are not involved in the dependence, 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 individual 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 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. A few of the MOV instructions are instructions that load immediate values into registers.

When we consider scalar basic blocks alone of any program class, the miscellaneous operations assume much greater significance, constituting close to 50% of all the operations. When considering an exclusive scalar unit, it is important to either efficiently execute or eliminate these operations. Several interacting factors are at play in determining both the presence and the runtime cost of the miscellaneous operations. We discuss some of these factors below, to shed light on the issue. If a separate scalar unit is designed for scalar blocks, it might be designed to have shallower computation pipelines in order to decrease latencies, thus decreasing the number of stalls in instruction issue and hence decreasing the opportunities for free issue of the miscellaneous operations. But, memory latency is still around 14 clock cycles for the CRAY X-MP and slightly larger for the CRAY Y-MP; furthermore, the current trend is an ever-increasing main memory latency. Hence sufficient opportunities could still exist for such free issue of the miscellaneous operations. Note that when the pipelining level is decreased, the life times of registers in terms of clock cycles is also decreased, and thus the pressure on the register sets will decrease due to any decrease in the pipeline levels of the functional units. However, the memory latency might be the dominating effect on register lifetimes.

For a separate scalar unit, on the other hand, we need to consider the effects of new issues. An exclusive data cache that does not have to deal with vectors is a viable proposition[Smith90]. In this situation, the resulting shorter average memory latency will further reduce the pressure on the registers (they will be tied up for shorter times while waiting for a LOAD to complete). Thus, many of the miscellaneous operations are likely to disappear. (We also note in passing that the data cache itself could be used as the backup registers.) However, a note of caution is in order. If both computation and memory pipelines are made shallow, and memory latency is reduced, the instruction issue stage has the potential of becoming a bottleneck, and even a few miscellaneous operations might not be desirable. In this situation, having additional primary registers would avoid the need for issuing the miscellaneous operations. Shallower pipelines imply a longer clock cycle, and hence the increase in register file access time (due to an increase in register file size) would be less of a concern. Of course, the effects of an increase of register space on the instruction format have to be considered: larger instructions are necessary to address the extra register space.

To bring the issue full circle, if shallow pipelines are used along with a data cache, a superscalar architecture could be viable to exploit instruction-level parallelism. Since multiple instructions will be issued simultaneously, the number of live registers will proportionally increase, thus increasing the pressure on the primary registers again. Thus, all the issues discussed above have to be studied simultaneously, to determine the appropriate size of the primary registers. We suspect that for shallow pipelines the clock cycle will be sufficiently large that an increase in the primary register set size will not adversely affect the clock cycle, thus solving the problem of spill instructions.

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 machines require the programmer to handle dependencies amongst vector instructions in software in certain situations, and hence the need for a CMR/mini-CMR instruction. We observe from the tables on the scalar basic blocks that while all the BLK_LD instructions occur in scalar basic blocks, quite a few of the BLK_ST instructions occur in vector basic blocks. These BLK_ST instructions are mostly used as mini CMR instructions as discussed above. (The CRAY Y-MP requires memory 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 compiler optimized 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.

The scalar basic blocks of both versions of the programs show equal presence of floating-point adds and floating-point multiplies. Furthermore, most of the scalar floating-point instructions occur in the scalar basic blocks rather than in the vector basic blocks (except for the floating-point adds in the hand-optimized codes). This distribution is unlike that of the other scalar instructions, many of which are used in vector blocks for setting up vector code execution.

Memory Operations

From table 4.9, we observe that memory operations are the single most frequent class of operations for the vector and moderately-vector classes of both versions of the 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 tables 4.1 and 4.2 that for the non-scalar benchmarks, usually more than 90% of the memory operations are executed in vector mode (V_LD and V_ST). Even for the scalar programs, a significant fraction (around 40%) of all memory operations are vectorized. 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.

Across the benchmark classes, 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 dyadic instructions in the architecture. The issue is complicated by the reuse of temporary results in registers.

For the vector programs, we notice that, although all floating-point computations are dyadic instructions, the memory operations are not three times as frequent as the floating-point operations, which would be the case if each computation required two memory loads and a memory store. This suggests temporary results are stored in registers are reused quite frequently.

When we consider the scalar basic blocks of all programs in isolation, the frequency of memory operations is the same as that of the floating-point, integer, or address operations. LOADs are still approximately twice as frequent as STOREs. In a separate scalar unit, memory latency becomes critical, and a data cache might be in order[Smith90]. However, a detailed study of access patterns in these scalar basic blocks is necessary to address this issue in more detail. For example, the backup scalar and address registers (64 each) might be quite effectively used as data caches for some access patterns, especially since they can be explicitly preloaded using the BLOCK_LOAD instruction.

Address Computation

Most of the address computation instructions seen 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.

The proportion of address computation instructions is obviously higher in scalar programs and in the scalar basic blocks of all programs, where memory references are scalar. Vector memory references implicitly do a significant amount of address computation, and have little need for a separate address unit that executes explicit address computation instructions.

We notice from the tables on the scalar basic blocks that not all of the address computation instructions are found in the scalar basic blocks. A significant fraction of these instructions are used in the vector blocks, to set up the work of the vector instructions.

Branch Instructions

Both the compiler-optimized and the hand-optimized versions of the benchmarks have very similar frequencies of branch operations. As expected, branch instructions are most frequent in scalar code. However, the branch frequency in the compiler optimized version (table 4.9) is much less than the usual 20% of all instructions or so reported for general-purpose programs [Hwu89, Hennes, McFarl86]. 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, need not cause bubbles in the pipeline since the branch destination is known at compile time itself. Tables 4.16 and 4.17 provide categorization of the branches into different varieties.

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 classes. 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 4.9 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

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 4.16: Percentage of Branches of Various Types

— *Compiler-Optimized Codes*

Benchmark Subclass	Branch Type			
	Uncond.	Subroutine Calls	Conditionals	
			Loop Ctrl.	Data-Dep.
Moderate	3.7	13.6	35.8	46.9
Vector	0.4	13.0	61.4	25.3

Table 4.17: Percentage of Branches of Various Types
— *Hand-Optimized Codes*

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 4.16 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.

For the moderately-vector programs of the hand-optimized codes (table 4.17), close to half the branches are data-dependent branches. However, the frequency of branches with respect to other operations is lower in these benchmarks, and hence they exhibit higher vectorization levels.

Tables 4.18 and 4.19 present categorization of the branches in the scalar basic blocks of the original and the hand-optimized codes. As expected, data dependent branches are the most common type of branches; however, loop-control branches are

Benchmark	Uncond.	Subr.Calls	Conditionals	
			Loop Ctrl.	Data-Dep.
ADM	2.9	25.4	40.7	31.0
ARC3D	0.0	3.9	49.9	46.3
BDNA	0.0	46.5	48.7	4.9
DYFESM	5.1	5.7	40.4	48.9
FLO52	0.9	2.0	42.2	55.0
MDG	1.2	12.7	46.5	39.6
MG3D	0.1	3.1	52.2	44.6
OCEAN	0.7	0.9	86.5	11.9
QCD	3.0	51.1	11.0	34.9
SPEC77	4.5	15.0	58.1	22.4
SPICE	19.8	16.6	4.7	58.9
TRACK	0.1	7.1	39.7	53.1
TRFD	0.0	3.3	33.7	63.1

Table 4.18: Percentage of Branches of Various Types in Scalar Blocks
— *Compiler-Optimized Codes*

Benchmark	Uncond.	Subr.Calls	Conditionals	
			Loop Ctrl.	Data-Dep.
ADM	4.3	18.5	25.6	51.6
ARC3D	0.0	3.3	50.8	45.9
BDNA	0.0	88.6	3.7	7.7
DYFESM	3.5	6.7	47.2	42.6
FLO52	0.3	1.3	53.5	44.8
MDG	0.0	20.6	45.8	33.6
MG3D	0.1	4.0	52.5	43.3
OCEAN	0.0	11.2	34.3	54.5
QCD	9.6	15.3	8.0	67.2
SPEC77	4.2	13.6	55.9	26.3
SPICE	5.7	4.6	43.8	45.9
TRACK	0.8	38.1	15.8	45.3
TRFD	0.0	0.0	49.9	50.1

Table 4.19: Percentage of Branches of Various Types in Scalar Blocks
— *Hand-Optimized Codes*

also quite significant in number. We also see a fair share of subroutine calls, with QCD and BDNA at the extreme, showing 50% of their branches to be subroutine calls. Unconditional branches are only a small fraction, except for the automatically compiled version of SPICE which has about 20% unconditional branches. The frequency of loop control branches suggests the presence of several loops with conditional branches or subroutine calls within their bodies that prevents their vectorization.

4.3. 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 [Hwu89, Hennes, McFarl86]. 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 and/or have equal impact on performance, the compiler could incorporate different techniques to tackle the two varieties of basic blocks.

Figure 4.1 presents the cumulative (dynamic) distribution of basic blocks in the three classes of the original 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 classes. 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 classes. 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 classes: 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[1] block size is about 14 instructions for vector programs, about 18 instructions for the moderately-vector pro-

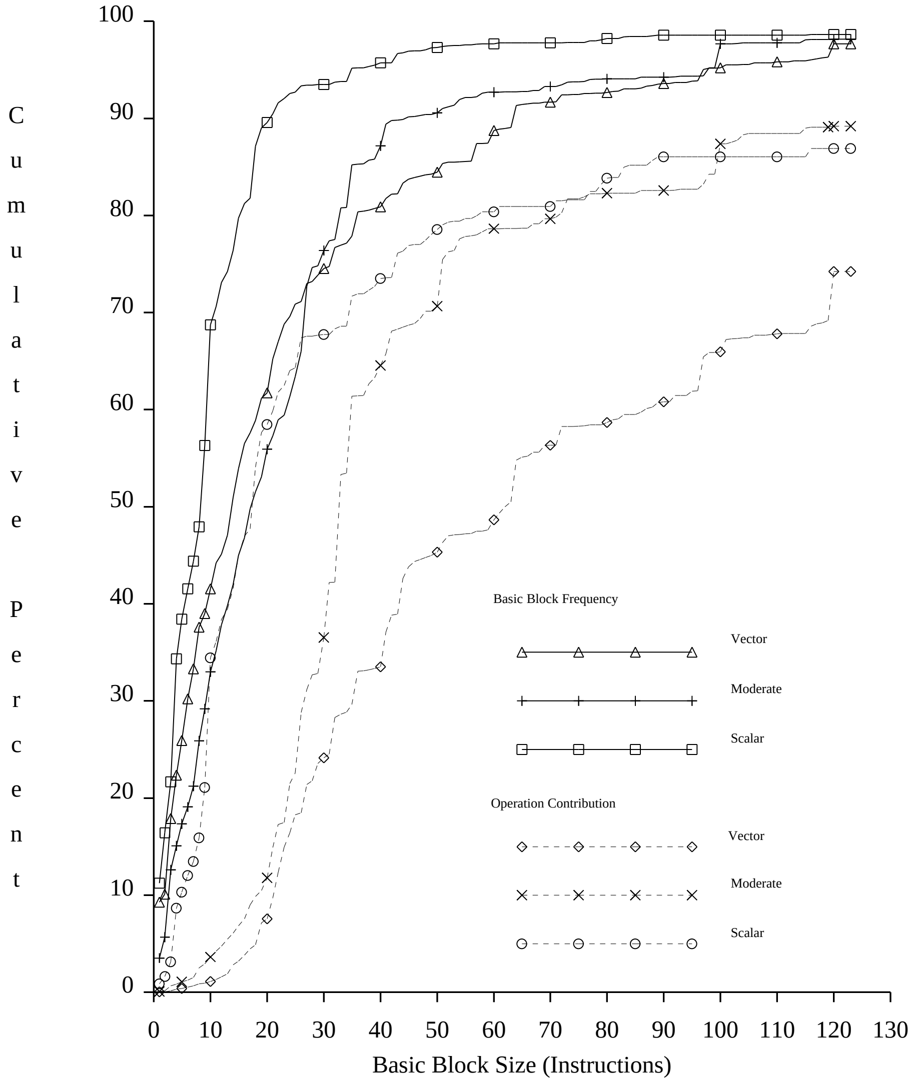
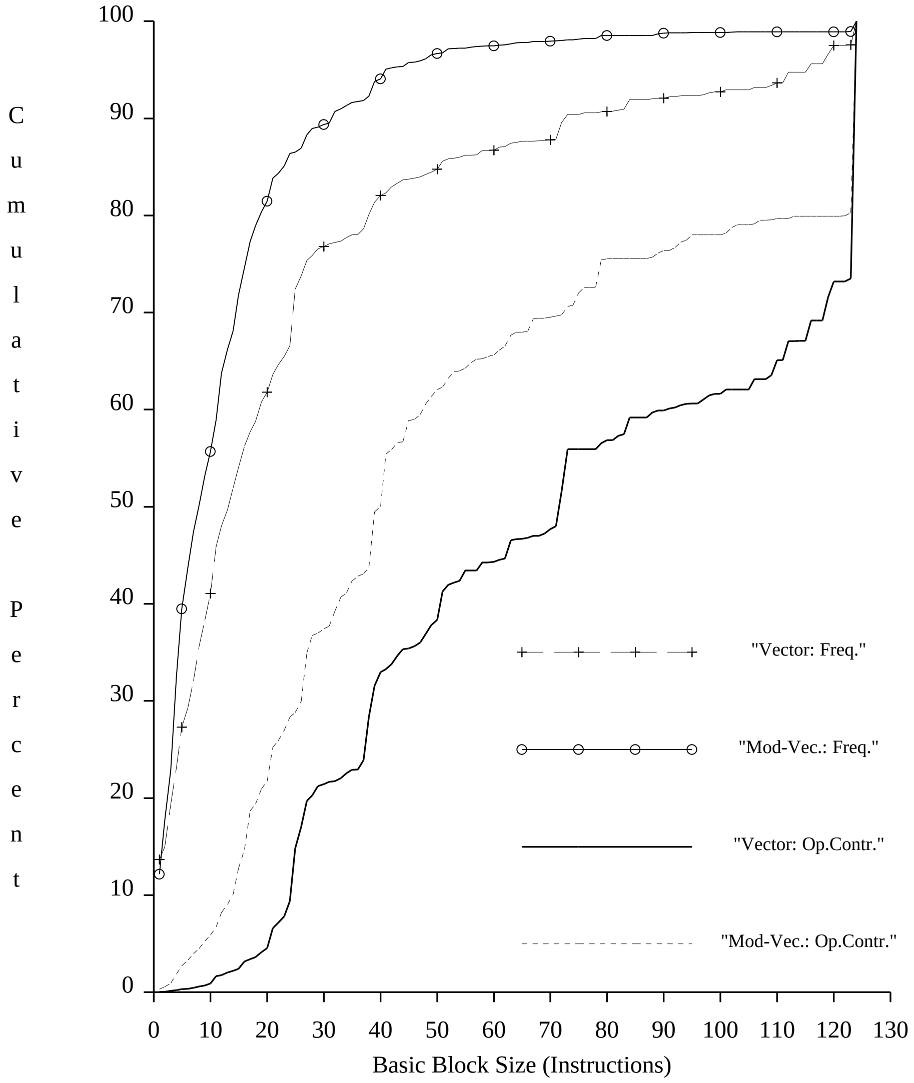


Figure 4.1: Basic Blocks in the Benchmarks

— *Compiler-Optimized Codes*

grams, and between 8 and 9 instructions for the scalar programs.

[1] 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



— *Hand Optimized Codes*

Figure 4.2: Basic Blocks in the Benchmarks

— *Hand Optimized Codes*

the 50% mark on the y-axis.

Figure 4.2 presents the same data for the two classes of the hand-optimized benchmarks. In the hand-optimized programs, 90% of the basic blocks of the moderately-vector programs range from 1 to about 30 instructions, while for vector programs they range from 1 to 70 instructions.

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 4.1 and Figure 4.2. For example, we observe a significant number of blocks less than 10 instructions in size, and a significant number larger than 100 instructions in size in the programs. 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 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 pipeline latencies of the CRAY Y-MP, eliminating several loop branches in the process and increasing basic block sizes. 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). 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 classes 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.

Figures 4.1 and 4.2 also 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 classes 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. For the compiler optimized programs, 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.

When we consider only the scalar basic blocks in the programs (figures 4.3 and 4.4), we see that most of them are between 1 and 20 instructions in size, except for the moderately-vector original programs, which have a larger spread of up to 40 instructions. Considering the fact that half the instructions in the scalar blocks are miscellaneous operations, eliminating them would result in a spread of block sizes between 1 and 10 instructions. Although the average block size would then be 5 instructions, very similar to that of general purpose (non-engineering) programs, we will have a good spread of basic block sizes between 1 and 10 instructions. In general purpose programs, on the other hand, a spread of basic block sizes is not usually expected, and the sizes of most blocks are expected to cluster around 5 instructions. Thus, we see larger scalar basic blocks in our scientific benchmarks than in general purpose programs. This suggests better code scheduling opportunities and probably more instruction level parallelism in these blocks compared to general purpose programs.

4.4. LIBRARY CALLS

Table 4.20 lists the number of user-routine and library-routine calls in the two versions of the benchmarks. Usually, subroutine calls are detrimental to performance since they prevent vectorization of the caller and also incur huge costs on the Cray machines in terms of saving/restoring a large number of registers. However, library routines are hand-tuned on the Cray machines to provide high performance, and hence they are beneficial to performance. We observe that hand-optimization has drastically reduced the number of subroutine calls in many programs. (The numbers for the two versions of DYFESM and FLO52 are not to be compared since the hand-optimized version was available to us only for a larger data set.) This could be achieved either via in-line expansion of subroutines or by rewriting the code. We note

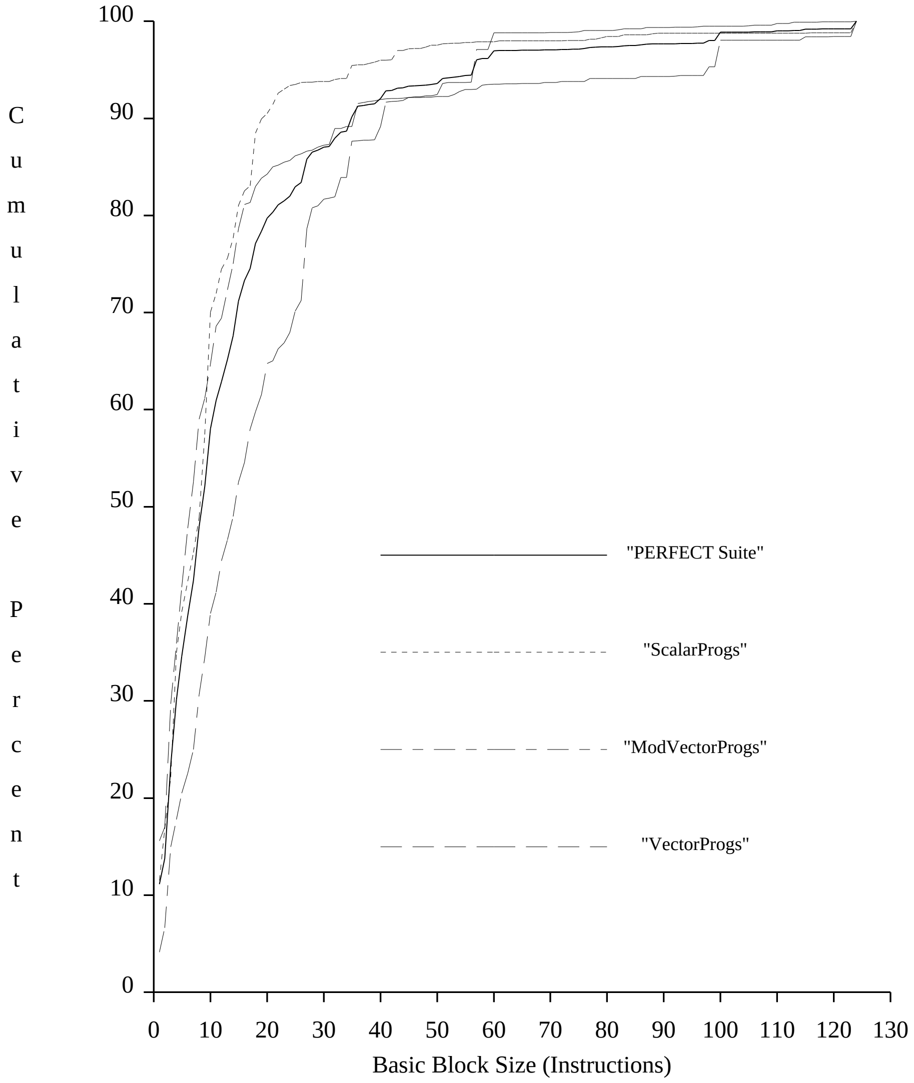


Figure 4.3: Scalar Basic Blocks in the Benchmarks

— *Compiler-Optimized Codes*

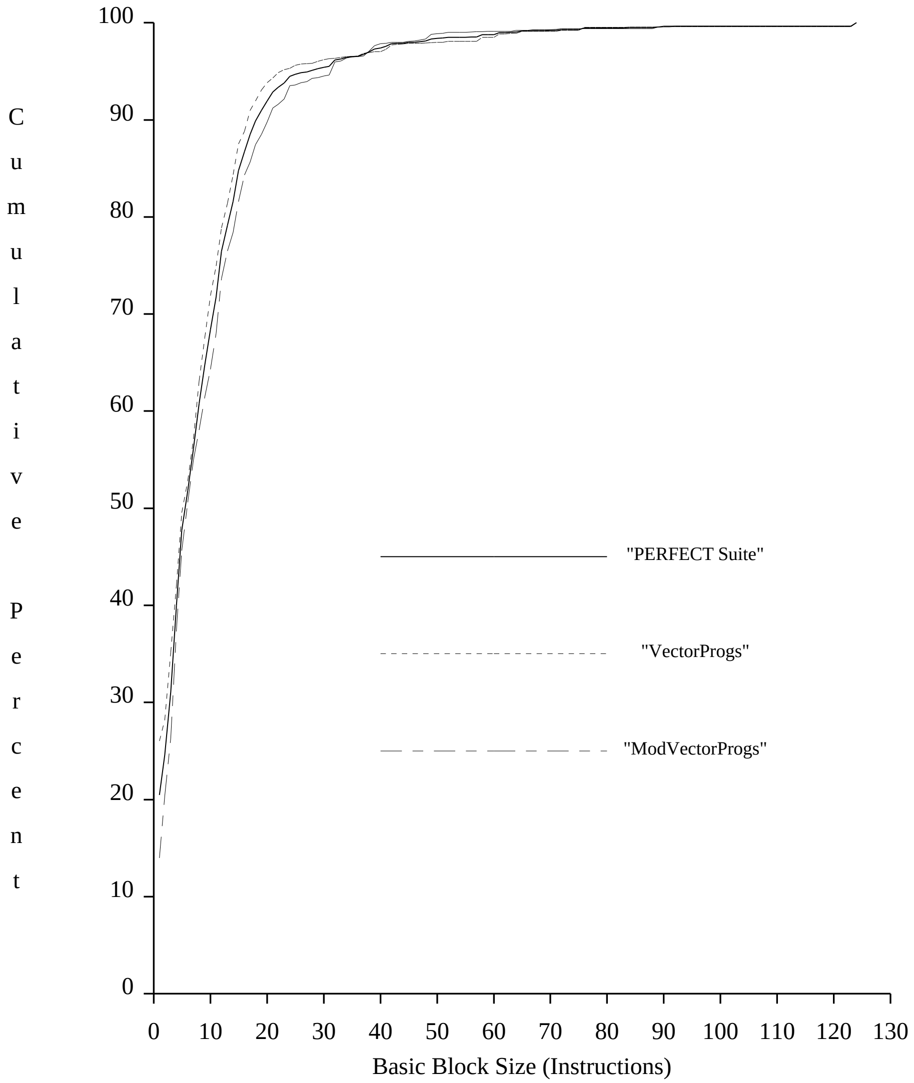


Figure 4.4: Scalar Basic Blocks in the Benchmarks

— *Hand-Optimized Codes*

that the subroutine calls in ADM and QCD are reduced to a very large extent. MDG, MG3D, and TRFD show marginal increases in user-routine calls, which might be due to program modularization.

Benchmark	User-Routines		Library Routines	
	Orig.	Hand-Opt.	Orig.	Hand-Opt.
ADM	2,760,607	178,002	6,046,461	966,474
ARC3D	24,005	303	927,850	868,632
BDNA	58	41	435,046	443,161
DYFESM	107,146	142,176	224,881	402,041
FLO52	10,668	10,670	59,196	59,245
MDG	121	219	5,926,880	2,261,764
MG3D	47,390	47,782	187,272	191,233
OCEAN	150,876	147,323	304,810	306,017
QCD	3,814,285	969,253	8,122,222	1,960,103
SPEC77	172,183	168,966	524,725	512,355
SPICE	1,485,177	51,501	2,215,681	126,727
TRACK	652,717	652,707	2,426,801	1,825,832
TRFD	16	17	814,474	174

Table 4.20: Subroutine Calls in the Benchmarks.

All the programs have many more library calls than calls to user routines. Scientific programs invoke several standard computation functions (such as square roots, etc.), and these are provided as optimized library routines on the CRAY Y-MP.

Also, we note that our benchmarks are written in FORTRAN; this could be a reason for the relatively few subroutines in the programs.

4.5. INSTRUCTION AND OPERATION ISSUE

Parallel instruction issue is the focus of much current research (the current work on superscalar and VLIW architectures). Parallelism in the issue stage and pipelining of the processor are roughly equivalent in exploiting fine-grain parallelism [Jouppi89, Sohi89]. A processor with deeply pipelined functional units has less need for parallelism in the issue stage [Sohi89]. We discussed the above issues in some detail in Chapter 2. In this section, we investigate the utilization of the instruction issue stage of the deeply-pipelined CRAY Y-MP processor by the benchmark programs. (Please refer to Chapter 3 for the functional unit latencies and a description of the processor).

Table 4.21 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 4.21 identifies the 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. The issue stage utilization is not very high overall, ranging between 10% and 43%. Table 4.22 presents the instruction issue rates for the hand-optimized programs, and we see very similar overall behavior. 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 limit of 1 instruction per cycle. For example, if the floating-point add unit has 7 pipeline stages, then 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 instruction issue rates than the other benchmarks. Vector instructions execute several operations, and hence fewer

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
BDNA	0.193	0.193	0.118	0.311
MDG	0.197	0.197	0.075	0.272
ARC3D	0.212	0.212	0.056	0.268
FLO52	0.220	0.220	0.066	0.286
DYFESM	0.335	0.335	0.124	0.459
ADM	0.355	0.355	0.139	0.494
TRFD	0.386	0.386	0.078	0.464
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 4.21: Instruction Issue Rate and Issue Stage Utilization
— *Compiler-Optimized Codes*

Program	Instruction Issue Rate	Issue Stage Utilization		
		by 1st parcel	by 2nd/3rd parcels	by all parcels
MG3D	0.110	0.110	0.024	0.134
BDNA	0.117	0.117	0.050	0.168
MDG	0.154	0.154	0.047	0.201
OCEAN	0.172	0.172	0.046	0.218
SPEC77	0.186	0.186	0.053	0.239
ARC3D	0.204	0.204	0.059	0.264
FLO52	0.274	0.274	0.075	0.350
QCD	0.224	0.224	0.237	0.462
SPICE	0.246	0.246	0.143	0.389
ADM	0.339	0.339	0.140	0.479
TRACK	0.359	0.359	0.178	0.537
DYFESM	0.368	0.368	0.162	0.530
TRFD	0.440	0.440	0.086	0.526

Table 4.22: Instruction Issue Rate and Issue Stage Utilization

— *Hand-Optimized Codes*

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. Tables 4.23 and 4.24 present very approximate estimates of operation issue rates for the user routines of the benchmarks. These operation issue rates are arrived at by the following approximate method: the execution of each static block is simulated *in isolation*, and the execution time in clock cycles for that block is calculated. Then, the execution time of each static basic block is multiplied by the frequency of execution of that block, to obtain an estimate of its contribution to program execution time. The sum of these estimates for all the blocks is used as the approximate execution time of the user routines of the program; since we know the operation count in the user routines, the operation issue rates are now calculated. We note that the above is very approximate (as we are ignoring or approximating cross-block dependences, memory bank conflicts, instruction cache misses, etc.). We present these estimates only to give a feel for the operation issue rates on the machine.

We see that the vector programs have higher average operation issue rates than the scalar programs, as expected. The scalar programs have operation issue rates between 0.4 and 0.6 operations per cycle, while most of the vector programs have between 2.0 and 2.5 operations issued per cycle. The moderately-vector programs lie in between in the operation issue rate distribution. We note that an operation issue rate of 2.5 for the vector programs suggests that, on the average, 2.5 vector instructions are in progress simultaneously with each issuing an operation every clock. Usually, there is much more parallelism among the vector instructions; we point out below some of the reasons for the limited parallelism seen in the execution of vector instructions.

An important reason for the low instruction and operation issue rates for the vector programs 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*. Empirical evidence to this effect is presented in [Tang88]. (Our approximate study also confirms the above; we do not present any data here since our studies are very approximate, with timing errors of up to 30% for the individual programs.) For example, the vector registers could be a bottleneck since the processor has only 8 vector registers. If each vector instruction needs 2 operand vector registers and 1 result vector register, only 2 vector instructions can be in progress simultaneously. (We note that there is a little more potential for parallel vector instruction execution on the Cray machines due to the presence of *vector chaining* which enables a dependent vector instruction to start execution at the time the first element of each of its operands is available. Also, vector LOADs and STOREs need just one register each.) 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 control- and 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. Thus, there is ample opportunity for the spill

Program	Operation Issue Rate
SPEC77	2.0
MG3D	2.0
FLO52	2.0
BDNA	2.0
ARC3D	2.0
MDG	1.6
TRFD	1.4
DYFESM	1.2
OCEAN	0.8
ADM	0.8
QCD	0.6
TRACK	0.4
SPICE	0.4

Table 4.23: Operation Issue Rate — *Compiler-Optimized Codes*

Program	Operation Issue Rate
SPEC77	2.5
BDNA	2.4
MG3D	2.2
FLO52	2.2
ARC3D	2.2
OCEAN	2.1
MDG	2.0
TRFD	1.8
SPICE	1.5
QCD	1.4
DYFESM	1.3
ADM	1.1
TRACK	1.0

Table 4.24: Operation Issue Rate — *Hand-Optimized Codes*

instructions to be issued for free during clock periods which would otherwise have been stalls for data dependences to be resolved; of course, this is possible only when the spill instructions themselves are not stalled due to dependences. 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 a separate scalar unit, especially one with shallow pipelines, this could be important since the instruction issue stage has the potential of being a bottleneck when the pipelining is decreased.

In conclusion, the issue stage does not appear to be a bottleneck, *on the average*, 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).

4.6. SUMMARY

We presented in this chapter a study of the single processor of the CRAY Y-MP using as benchmarks programs from the PERFECT Club set. We compared the instruction-level behaviors of compiler optimized benchmarks, hand-optimized benchmarks, and just the scalar basic blocks of both of these versions. We characterized the processor by studying program vectorization levels, instruction usage patterns, and basic block sizes for our benchmarks. We also investigated user-routine and library calls by both versions of the benchmarks. Finally, we studied the instruction and operation issue rates of the benchmarks. We present below some of the important conclusions of the above studies.

4.6.1. Program Vectorization

The PERFECT Club benchmarks exhibit a wide range of vectorization, in contrast to the usually expected high level of vectorization of scientific programs. Anywhere between 4% and 96% of the operations of the programs are vectorized when vectorization is carried out by the state-of-the-art Cray Research, Inc. FORTRAN compiler. Upon hand-optimization of the benchmarks, the vectorization level improves considerably for many programs, with the percentage of operations vectorized now ranging from 55% to 97%. The hand-optimized version of the benchmarks studied is the version that won the 1990 Gordon Bell-PERFECT award for the fastest supercomputer applications, and was optimized by a team of Cray Research, Inc. programmers. Two of the benchmark programs, SPICE and QCD, show as much as 70% additional vectorization thanks to hand optimizations.

Thus, the performance attainable using current state-of-the-art vectorizing compilers is much less than the performance available via manual optimizations. Knowledge of the underlying vector machine on the part of the programmer is still very necessary to harness the full power of the machine.

4.6.2. Instruction and Operation Counts

We classify programs into vector, moderately-vector, and scalar classes when studying their characteristics. While the compiler-optimized version of the benchmarks has programs of all three classes, hand-optimization moves all programs into either the moderately-vector or the vector class, thus completely eliminating the scalar class of programs. Overall, the proportions of instructions and operations of each class of programs is not affected by whether the programs were compiler-optimized or hand-optimized. Thus, for example, the moderately-vectorized programs of both the compiler-optimized and the hand-optimized versions have generally similar characteristics.

The vector programs are predominantly floating-point operations and memory accesses, with these forming close to 90% of all operations executed. The moderately-vector and the scalar programs, with their lower vectorization levels, have significant proportions of integer as well as address-computation operations (which, while being integer calculations, are distinguished from other integer operations) in addition to floating-point and memory operations. Very interestingly, spill operations that move values between the scalar primary register sets and the corresponding backup register sets and move operations that move values between the two scalar primary register sets together constitute the largest fraction of operations in scalar programs as well as in the non-vectorized basic blocks of the moderately-vector and vector programs. For example, these miscellaneous operations, which are specific to the Cray machines, constitute close to 40% of all operations of the scalar programs of the compiler-optimized benchmarks. While the spill instructions can be eliminated by using a larger register set, the impact on clock cycle, instruction format, etc., have to be considered when increasing the register set size.

While almost all the memory operations of the non-scalar programs are vectorized, a very significant fraction of the memory operations of even the scalar programs are vectorized. For example, though only about 14% of all operations of TRACK are vectorized by the compiler, close to 50% of its memory operations are vectorized. Thus, many of the memory references are predictable, and large memory bandwidth is desirable for these programs.

Branches form only about 6% of the operations of the scalar programs, and are less than 2% of all operations in the other two program classes. Eliminating the spill instructions from the scalar programs will increase the proportion of branches, but the frequency is still much less than the 20% reported for non-scientific programs. In the scalar programs close to half the branches are data-dependent branches, and about a quarter are subroutine calls. The significant presence of loop control branches in scalar blocks suggests the presence of loops with conditional branches and/or subroutine calls that make them non-vectorizable. In the moderately-vector and the vector programs, the majority of the branches are expectedly loop-control branches.

4.6.3. Basic Blocks

The sizes of basic blocks are important in the choice of compiler techniques for code optimization and scheduling. Basic blocks in non-scientific applications are reported to be around 5 instructions long, and heuristic compiler algorithms tuned to this size are hence used in compilers. We find that in our benchmarks small blocks as well as large blocks that are more than 100 instructions in size are frequent. Furthermore, the large blocks contain a number of vector instructions and hence contribute significantly to the dynamic operation count and thus to program execution time. For example, blocks larger than 125 instructions contribute more than 12% of the operations for the scalar programs and more than 25% of the operations of the vector programs in the compiler-optimized version of the benchmarks. Hence, code optimization and scheduling techniques geared toward large blocks are desirable, in addition to ones geared toward small blocks. Apart from the vectorized blocks, the blocks in the scalar programs as well as the scalar blocks in the non-scalar programs have median block sizes between 8 and 10 instructions, but are widely distributed in the range of 1 to 20 instructions and beyond. In fact, for the compiler-optimized benchmarks, 10% of the scalar basic blocks of all program classes are at least 20 instructions in size.

4.6.4. Instruction and Operation Issue Rates

The instruction issue rates range between 0.1 instructions per cycle and 0.45 instructions per cycle for both the hand-optimized and the compiler-optimized versions of the programs. Since each vector instruction issues multiple operations, the operation issue rates are expectedly higher, ranging between 0.4 and 2 operations per cycle for the compiler-optimized benchmarks and between 1.0 and 2.5 operations per cycle for the hand-optimized benchmarks. The deeply pipelined functional units and the long memory latency result in much of the operation-level parallelism being exploited by pipelining and hence we see comparatively less parallelism at the issue stage. Furthermore, the quantity of vector resources in the Cray machines limits the exploitation of the parallelism between vector instructions, as has been reported in the literature.

This chapter has mainly focused on the vector program as a whole; the next chapter deals with issues that are of importance specifically to the scalar portions of vector programs.

Chapter 5

CHARACTERIZATION OF SCALAR BASIC BLOCKS

5.1. INTRODUCTION

The scalar code in vectorized programs has characteristics and consequently resource requirements that are very different from those of the vectorized blocks of the vectorized programs as well as from those of non-vectorized programs. We discussed this issue and its implications in detail in chapter 3. As discussed therein, we are interested in understanding the characteristics and resource requirements of scalar code in order to aid the design of a separate scalar processing unit targeted specifically at scalar code. Towards this end, we studied the instruction mix and the sizes of the scalar basic blocks in the previous chapter, and compared them with those of the vector blocks. In this chapter, we focus on several other issues specifically important to the fast execution of scalar basic blocks. In section 5.2, we discuss inter-instruction data dependencies. In section 5.3, we discuss issues related to quick branch execution. In section 5.4, we discuss functional-unit pipeline utilization. In each of these sections, we discuss the importance of the issues addressed, present and analyze relevant data collected, and draw conclusions about their implications on scalar processor design.

5.2. DATA DEPENDENCIES

The data dependencies in a program determine the amount of instruction-level parallelism in the program, the amount of communication between various types of instructions, etc. Code segments that are not vectorized usually have a large amount of data- (and control-) dependencies, which is the reason they are not vectorized. The dependencies seen in these code segments are critical to the design of the scalar processor. Understanding the nature of these dependencies enables one to design faster scalar units, for example by providing more communication paths between frequently-communicating functional units. Furthermore, the data-dependencies can be exploited to provide faster instruction execution. For example, the IBM RS6000 [Oehler90] provides a compound multiply-add functional unit that executes (y = a*b + c) faster than executing (x = a*b) followed by (y = x + c). Of course, such a functional unit would be cost-effective only if a sufficient amount of such dependencies between multiply instructions and add instructions exist in the programs. Also, when two simple instructions are combined into a compound instruction, the result of the first simple instruction may no longer be available for use by other instructions. This imposes additional limitations on the use of compound functional units, as will be discussed shortly. Thus, characterizing the dependencies in programs is key to making several tradeoffs and design choices in the processor.

In this chapter we examine instruction dependencies within individual scalar basic blocks. It would be interesting to also examine such dependencies across basic blocks. However, our methodology, discussed in detail in chapter 3, does not provide us such data. We note that to exploit dependencies across basic blocks, it is essential that both branch-prediction techniques and recovery mechanisms be available, as for example in a system that uses trace-scheduling[Fisher81]. For example, suppose instruction "x" of basic block A feeds its output to instruction "y" of basic block B which is a potential successor of block A. If we were to replace instruction "x" with the compound instruction "xy", and if branch-prediction were to go wrong and some basic block C is executed instead of block B, then block C has to have appropriate compensation code to undo "xy" and redo "x". Thus, exploiting dependencies within a basic block is a more-easily implementable first step in exploiting dependencies.

We also do not study memory dependencies (i.e., STORE -> LOAD dependencies), since we do not deal with address traces in our studies. Such dependencies exist due to two reasons: (i) the memory address accessed by the operations, and hence the exact data-dependencies between the operations, are not known at compile-time, resulting in the value that causes the memory dependence not being assigned to a register, or (ii) when such dependencies are known at compile-time, the compiler is unable to allocate the value being accessed to a register due to either register shortage or to limitations of the register allocation algorithm. In the latter case, if an improved compiler were to allocate the value to a register we might see additional exploitable dependencies between computation instructions. For a given compiler, however, ignoring memory dependencies does not affect a study such as ours.

In our study we eliminate the Cray-specific spill instructions so as to obtain more general results. Spill instructions are eliminated in the following manner. If the result of instruction "x" is relocated by spill instructions and then used by instruction "y", this is accounted for as a direct dependence of instruction "y" on instruction "x". Thus, we are able to study a scenario where the effect of effect of spill instructions, which are present due to a shortage of primary registers, has been eliminated.

Before we examine the frequency of various dependencies, let us consider some factors that influence the effectiveness of exploiting data-dependencies via compound functional units. Suppose an instruction has *multiple* dependents, either within the same basic block as the instruction or otherwise. If the instruction is incorporated into a compound instruction for a compound functional unit, all the other dependents of the instruction have to be provided the result of this instruction. This can be accomplished in several ways, and the appropriate choice is dependent on the particular number of dependencies, functional unit latencies, etc. For example, each dependent of the instruction could be replaced by a compound instruction. Or, some of the dependents could be replaced by compound instructions, and the instruction also executed separately once to provide its result to the other (non-combined) dependents. This approach might be effective for exploiting the dependencies of an instruction within a basic block even when it has additional dependents in other blocks. In some implementations, the result of the instruction might also be available for free from the compound functional unit, thus eliminating the need for executing the instruction separately to take care of its non-combined dependents. Of course, the simplest approach overall is to combine an instruction into a compound instruction if it has exactly one dependent and that dependent is in the same basic block. The compiler can play a significant part in such exploitation of dependencies between instructions.

To illustrate some of the above points, consider the following two simple expressions:

m = b*c + d
n = b*c + e

Suppose the multiply operation takes 3 clocks and the add operation takes 2 clocks, but the compound operation (p*q+r) takes only 3 clocks. A naive compiler might compute x = b*c followed by m = x+d and n = x+e. This takes (3+2+2)=7 clocks, assuming the three operations are executed strictly in sequence, for simplicity. However, using the compound instruction (s=p*q+r) for both m and n results in an execution time of only 6 clock cycles, again assuming strictly sequential execution, for simplicity. Clearly, the appropriate choice of expression formation is dependent on functional unit latencies, scheduling considerations, etc.

We note that the compiler might be able to move instructions across loop iterations to exploit compound functional units. This could be quite simple and very effective, since the loop control branch is taken most of the time, and for many cases only very simple compensation code needs to be inserted into the post-loop basic block to correct for the error at the end of the loop. Furthermore, when complex expressions are involved within a basic block, the parsing of the expressions could play a role in the dependencies that are seen between instructions.

Thus, although our study presents an initial overview of the exploitable dependencies in the programs, code compiled appropriately for specific proposed mechanisms have to be studied before drawing firm conclusions.

Tables 5.1, 5.2, 5.3, 5.4, & 5.5 present statistics of data dependencies in both the hand-optimized and the compiler-optimized versions of our benchmarks. The tables are read as follows. The "Isolated" row of the tables presents the percentage of instructions in the programs whose results are not consumed within the same basic block. For each instruction type, the corresponding column lists the distribution (percent) of all the dependents of that instruction type, with the column summing to 100%. For example, from Table 5.1, 14.3% of all the floating-point add instructions of the scalar benchmarks feed no instructions within the same basic block that they occur in. Thus, these isolated instructions can not be used to exploit a compound multiply-add functional unit without branch-prediction and recovery support. On the other

	LD	FP-MUL	FP-ADD	S-ADD	A-ADD	A-MUL	S-LOGIC	S-SHIFT	RECIP	MOV	POP-LZC
LD	0.50	-	-	-	24.25	42.72	-	-	-	19.55	-
ST	3.69	9.25	21.43	31.16	5.63	7.18	3.66	-	-	11.80	-
FP_MUL	35.21	40.79	25.66	-	-	-	1.62	33.40	99.44	1.64	-
FP_ADD	21.47	49.11	35.66	8.74	-	-	4.38	2.09	-	6.46	-
S_ADD	19.29	0.21	2.68	10.72	-	-	36.51	58.61	0.56	12.54	-
A_ADD	3.79	-	-	-	7.98	50.10	-	-	-	29.81	-
A_MUL	0.02	-	-	-	-	-	-	-	-	2.90	-
S_LOGIC	11.50	0.10	2.22	14.14	-	-	19.17	1.38	-	0.70	-
BR	1.29	0.21	9.76	10.84	32.05	-	30.83	1.27	-	-	-
BLK_LD	-	-	-	-	13.84	-	-	-	-	8.05	-
BLK_ST	-	-	-	-	13.95	-	-	-	-	5.33	-
S_SHIFT	0.17	-	1.83	1.76	-	-	0.75	3.24	-	0.43	66.67
RECIPR	0.84	0.23	0.77	-	-	-	0.21	-	-	0.02	-
MOV	2.24	0.11	-	22.64	2.30	-	2.34	-	-	0.77	33.33
POP_LZC	-	-	-	-	-	-	0.53	-	-	-	-
TOTAL	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Isolated(%)	8.73	9.13	14.33	9.67	28.93	-	17.17	9.10	9.60	35.23	-

Table 5.1: Data Dependencies in Scalar Programs

— *Compiler Optimized Codes*

	LD	FP-MUL	FP-ADD	S-ADD	A-ADD	A-MUL	S-LOGIC	S-SHIFT	RECIP	MOV	POP-LZC
LD	0.43	-	-	-	34.57	-	-	-	-	2.60	0.05
ST	4.84	3.67	35.00	4.71	7.27	-	0.68	0.05	0.04	10.39	-
FP_MUL	24.71	31.78	34.27	0.01	-	-	0.33	0.75	99.79	1.07	-
FP_ADD	33.74	62.78	12.64	6.98	-	-	2.21	1.81	0.14	6.30	-
S_ADD	6.63	1.24	10.86	28.18	-	-	33.97	65.18	0.04	18.91	-
A_ADD	3.10	-	-	-	20.70	60.04	-	-	-	53.30	-
A_MUL	0.63	-	-	-	0.09	1.80	-	-	-	2.98	-
S_LOGIC	22.62	0.26	2.70	16.45	-	-	27.36	3.98	-	0.46	-
BR	0.09	-	0.71	5.75	19.94	-	10.79	25.02	-	-	-
BLK_LD	-	-	-	-	1.76	-	-	-	-	1.47	-
BLK_ST	-	-	-	-	1.85	-	-	-	-	1.00	-
S_SHIFT	0.02	-	0.47	6.64	-	-	3.69	3.20	-	0.54	99.95
RECIPR	0.16	0.19	3.08	-	-	-	0.05	-	-	0.04	-
MOV	3.04	0.08	0.27	31.10	13.82	38.17	20.02	-	-	0.95	-
POP_LZC	0.01	-	-	0.18	-	-	0.88	-	-	-	-
TOTAL	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Isolated(%)	33.63	3.55	29.63	24.65	49.03	2.40	24.63	13.73	0.05	36.13	-

Table 5.2: Data Dependencies in Moderately-Vector Programs

— Compiler Optimized Codes

	LD	FP- MUL	FP- ADD	S- ADD	A- ADD	A- MUL	S- LOGIC	S- SHIFT	RECIP	MOV	POP- LZC
LD	0.23	-	-	-	30.93	-	-	-	-	3.93	3.52
ST	13.03	9.84	27.64	18.88	8.66	-	10.39	9.32	-	5.91	-
FP_MUL	25.26	19.81	35.34	-	-	-	0.05	3.87	98.13	0.19	-
FP_ADD	33.94	65.99	17.68	6.72	-	-	0.25	4.09	0.80	1.66	-
S_ADD	13.40	0.27	8.92	17.61	-	-	19.56	53.51	-	18.77	-
A_ADD	2.97	-	-	-	11.75	66.30	-	-	-	64.64	-
A_MUL	1.28	-	-	-	0.01	-	-	-	-	1.99	-
S_LOGIC	2.95	2.96	0.74	3.81	-	-	22.98	8.75	-	0.26	-
BR	1.57	0.45	4.87	21.92	32.48	-	17.71	12.10	-	-	-
BLK_LD	-	-	-	-	1.02	-	-	-	-	0.71	-
BLK_ST	-	-	-	-	1.06	-	-	-	-	0.45	-
S_SHIFT	-	-	-	0.92	-	-	0.08	5.75	-	0.18	80.74
RECIPR	1.78	0.16	2.74	-	-	-	-	-	0.07	-	-
MOV	3.56	0.52	2.06	30.14	14.08	33.70	28.98	2.13	1.00	1.30	15.74
POP_LZC	0.05	-	-	-	-	-	-	0.48	-	-	-
TOTAL	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Isolated(%)	18.72	16.33	10.43	46.45	54.65	0.08	33.78	14.33	-	47.03	-

Table 5.3: Data Dependencies in Vector Programs

— *Compiler Optimized Codes*

	LD	FP-MUL	FP-ADD	S-ADD	A-ADD	A-MUL	S-LOGIC	S-SHIFT	RECIP	MOV	POP-LZC
LD	0.42	-	-	-	25.03	9.36	-	-	-	6.18	0.25
ST	12.81	9.14	34.38	15.88	8.18	6.68	5.52	0.14	-	9.21	-
FP_MUL	18.85	37.96	30.62	-	-	-	3.13	0.71	94.15	0.59	-
FP_ADD	20.43	48.70	13.42	2.96	-	-	2.33	3.36	5.40	2.25	-
S_ADD	20.97	-	10.18	17.08	-	-	26.65	84.21	-	10.83	-
A_ADD	8.14	-	-	-	18.61	81.18	-	-	-	50.50	-
A_MUL	1.58	-	-	-	-	-	-	-	-	2.88	-
S_LOGIC	7.88	3.16	0.67	10.84	-	-	21.61	7.43	-	1.45	-
BR	2.46	0.08	3.02	31.47	13.88	-	19.33	4.04	-	-	-
BLK_LD	-	-	-	-	7.82	-	-	-	-	5.49	-
BLK_ST	-	-	-	-	8.01	-	-	-	-	6.29	-
S_SHIFT	0.17	-	2.38	0.47	-	-	1.00	-	-	1.25	78.68
RECIPR	0.46	0.51	5.11	-	-	-	0.20	-	-	-	-
MOV	5.73	0.43	0.21	21.30	18.46	2.78	19.80	0.10	0.45	3.07	21.07
POP_LZC	0.09	-	-	-	-	-	0.41	-	-	-	-
TOTAL	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Isolated(%)	33.47	6.90	6.98	30.72	34.13	0.22	28.77	24.57	0.23	34.12	0.97

Table 5.4: Data Dependencies in Moderately-Vector Programs

— *Hand-Optimized Codes*

	LD	FP-MUL	FP-ADD	S-ADD	A-ADD	A-MUL	S-LOGIC	S-SHIFT	RECIP	MOV	POP-LZC
LD	0.26	-	-	-	16.71	-	-	-	-	8.76	0.62
ST	6.92	32.24	28.52	23.93	8.01	-	10.82	1.02	-	12.99	-
FP_MUL	27.75	25.26	41.81	0.01	-	-	0.88	1.07	99.45	1.88	-
FP_ADD	24.65	38.82	9.30	8.93	-	-	1.24	3.05	0.19	3.98	-
S_ADD	16.06	0.22	10.96	15.05	-	-	17.25	61.73	0.18	14.55	-
A_ADD	9.22	-	-	-	17.79	91.62	-	-	-	48.92	-
A_MUL	1.28	-	-	-	-	-	-	-	-	1.85	-
S_LOGIC	3.49	0.44	1.79	8.00	-	-	23.76	6.65	-	0.64	-
BR	1.12	1.58	1.05	9.57	25.60	-	18.98	25.31	-	-	-
BLK_LD	-	-	-	-	1.18	-	-	-	-	1.62	-
BLK_ST	-	-	-	-	1.20	-	-	-	-	1.66	-
S_SHIFT	0.05	-	1.48	1.74	-	-	0.06	0.73	-	0.43	93.04
RECIPR	3.74	1.43	5.08	-	-	-	-	-	-	-	-
MOV	5.47	0.01	-	32.77	29.50	8.38	27.00	0.15	0.17	2.70	6.35
POP_LZC	-	-	-	-	-	-	-	0.29	-	-	-
TOTAL	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Isolated(%)	24.29	18.63	23.96	38.56	44.91	4.41	29.60	25.94	6.84	46.80	-

Table 5.5: Data Dependencies in Vector Programs

— Hand-Optimized Codes

hand, from the same table we see that no address-multiply (AMUL) instructions of the scalar programs are isolated; each AMUL instruction of the scalar programs feeds at least one instruction within the same basic block.

Tables 5.6 and 5.7 present the average fanout of the results of various instructions for our benchmarks. These are fanouts within the basic block of the instruction; fanout beyond the basic block are not accounted for. We observe that the average fanout for the floating-point instructions is larger than 1 instruction on the average. For example, with an average fanout of 1.25 instructions, one out every five instructions will have multiple dependents and hence more involved mechanisms, as discussed above, are necessary to use that instruction in compound functional units. Note that both the dependence tables and the fanout tables present approximate figures in this respect since they ignore any additional dependencies the instructions might have across blocks.

Now let us examine in detail the dependencies of instructions of each instruction type. The dependencies of all instructions of each instruction type are listed in a column in each of the tables 5.1 through 5.5, with each column summing up to 100%. If an instruction has multiple dependents within its basic block, each of the dependencies is accounted for when computing the columns. In the discussion below, we term an instruction to be non-isolated if it has at least one dependent within its basic block.

A significant fraction of the instructions fed by address adds are, naturally, scalar memory operations. We note that in some of these memory operations the final

Inst.	Average Fanout (insts.)		
Type	Scalar	Mod-Vec.	Vector
LD	1.50	1.38	1.25
FP_MUL	1.60	1.22	1.13
FP_ADD	1.21	1.19	1.17
S_ADD	1.38	1.32	1.23
A_ADD	1.26	1.47	2.21
A_MUL	1.16	1.36	1.92
S_LOGIC	1.15	1.40	1.39
S_SHIFT	1.09	1.14	1.32
RECIPR	2.32	2.12	2.09
MOV	1.85	1.94	1.60
POP_LZC	1.00	1.00	1.21

Table 5.6: Average Fanout of Various Instructions

— Compiler Optimized Codes

Inst. Type	Average Fanout (insts.)	
	Mod-Vec.	Vector
LD	1.49	1.32
FP_MUL	1.16	1.18
FP_ADD	1.24	1.23
S_ADD	1.18	1.09
A_ADD	1.27	1.39
A_MUL	1.64	1.20
S_LOGIC	1.18	1.32
S_SHIFT	1.06	1.06
RECIPR	2.23	2.05
MOV	1.50	1.65
POP_LZC	1.06	1.01

Table 5.7: Average Fanout of Various Instructions

— *Hand-Optimized Codes*

memory address is obtained by adding an immediate value to the specified address register. One might considering exploiting this dependence between the explicit and implicit address calculation. However, additional complications such as having to assign a register to the immediate value and having to issue an additional instruction to move the value to the register might negate any advantages of exploiting the dependence. Another significant fraction of the dependents of address adds are MOVE instructions which move the result of the address adds to the scalar registers. This is because the address unit lacks some amount of functionality (as discussed in the previous chapter), and data has to be moved to the scalar unit to carry out those functions. We also see that LOADs are more dependent on address computations than STOREs. Finally, address adds are used on the Cray machines to increment loop counters. Loop-control branches testing the loop counters before branching results in a significant fraction of the dependents of address adds being branches.

From Table 5.1, for example, we see that 50.1% of the non-isolated address multiply instructions feed address add instructions. A significant fraction of the address multiplication instructions of all benchmark classes feed address adds. However, there are few address multiplication instructions in the programs, as shown in the previous chapter. Hence a combined multiply-add address functional unit may not be worthwhile.

A significant fraction of floating-point multiplies feed floating-point adds, and vice-versa. Also, a significant fraction of floating-point multiplies feed other floating-point multiplies. The same is true for floating-point adds, except that the fraction is smaller. Thus, one might conclude that various compound floating-point add-multiply units have the potential to be very effective. However, two issues need to be considered. First, up to 25% of the floating-point instructions could have no dependents within the same basic block, as seen from the tables. Second, we observe from tables 5.6 and 5.7 that the average fanout for the floating-point instructions is significantly greater that 1, thus necessitating some "smarts" in the compiler to exploit the dependencies.

The results of most of the remainder of the floating-point instructions are just stored back in memory. For the scalar programs alone, a good fraction (9.76%) of the floating-point adds determine branch conditions.

Upon examining the scalar add, shift, and logic instructions, we find similarly frequent inter-dependencies which might suggest the desirability of compound functional units. However, the same arguments presented above for the floating-point units apply here.

Let us now consider memory LOAD instructions. The highest fraction of LOAD instructions that have no dependents within that block is around 33%. The higher this fraction the lesser the impact of the relatively long memory latency on program execution speed. This is due to the fact that if the instructions that are dependent on LOADs are in other basic blocks (especially those that are "farther away" in the dynamic trace of basic blocks) there will be more independent instructions available to cover the LOAD latency with parallel work. We note that the Cray compiler attempts to move LOADs across loop-iterations to hide the relatively-long memory latency, thus implementing a restricted form of software pipelining[Lam88], for the memory operations alone. The fact that the majority of the LOAD instructions are not of this type despite this compiler optimization indicates that memory latency might play a large role in slowing down scalar code execution. This is especially true since the scalar blocks are relatively short when compared to memory latency, thus having fewer instructions available for issue during the latency of the load.

Most of the dependents of LOADs are naturally computation instructions. However, some of them are other LOADs as well as address computation instructions, suggesting the indirect accessing of memory (akin to pointer-chasing in C programs for example). Several of them are STOREs, suggesting data relocation in memory and also indirect memory accesses.

Finally we see a small fraction of the dependents of LOADs being branches. This is again significant for execution speed. Considerable execution overlap of instructions of different basic blocks can be achieved when the branches of the basic block are dependent on only a small portion of the instructions of the basic block. Suppose the branch instruction of the basic block can be issued while several of the instructions of the basic block are still being executed (though they were issued prior to the branch), or even when some instructions of the block are yet to be issued. One the branch instruction is executed, the successor basic block is determined. Now instructions from the subsequent basic block can also be issued, and these will overlap with those instructions of this block that are still in flight. However, to achieve this, the instructions of a basic block that affect the branch condition need to be few in number and have short latencies. If a LOAD were to affect the branch condition, inter-block parallelism would be hampered on a Cray machine due to the long memory latency.

Table 5.8 shows the distribution of instructions that play a role in determining branch conditions. Specifically, these are instructions, within the same basic block as the branch, that occur in the dependence chain that finally produces the branch condition. As discussed before, branches use registers S0/A0 to hold conditions, with A0 being used mostly for loop-control and S0 mostly for data-dependent branches. We see that LOADs form a significant fraction (13% and 16%) of these instructions for S0, and to that extent limit execution parallelism as discussed above. Among the other instructions that affect S0, the relatively short latency scalar adds and scalar logic instructions, together with spills and moves, form almost all of the instructions. We note that the long latency floating-point instructions form about 8% of the instructions that play a role in determining the branch condition. All the above instruction types affect S0 since the branches are data-dependent branches which test values involved in the computation work of the program. On the other hand, almost all the instructions that affect A0 are just address adds and spills, since most of these branches just test the loop counter which is maintained in a register. Hence these branches can be executed relatively quickly compared to the execution time of the basic blocks containing them.

A related issue is the latency of the final instruction in the dependence chain that produces the branch condition. For example, if a scalar add instruction produces the branch condition, it might be possible to obtain the branch condition by just determining the sign bit of the result of the add instruction instead of waiting for the entire result. For example, in a floating-point unit the sign bit can be calculated much more easily and much more quickly than the entire result. Thus the branch does not have to be stalled for the entire latency of the add instruction. Table 5.9 shows the instructions that produce the final branch condition. For branches based on the value in register A0, half of such instructions are address adds, while the rest are MOVs and SPILLs. Obtaining the sign bit of the result quickly from the address adder will hence decrease the waiting time of the loop control branches, and will be worthwhile. For condition register S0, the scalar logic unit produces a large fraction of the final branch condition; however, the latency of this unit is just one clock, and hence there might not be much

INST.	S0		A0	
	Original (%)	Hand-Opt. (%)	Original (%)	Hand-Opt (%)
LD	16.26%	13.26%	0.23%	1.60%
FP_MUL	2.85%	4.03%	-	-
FP_ADD	5.29%	4.70%	-	-
S_ADD	13.77%	15.68%	-	-
A_ADD	0.68%	0.38%	75.67%	58.70%
A_MUL	-	-	-	-
A_SPILL	0.26%	1.07%	23.86%	31.90%
S_SPILL	20.84%	17.74%	-	-
S_LOGIC	29.86%	33.18%	-	-
S_SHIFT	2.10%	2.19%	-	-
RECIPR	0.61%	0.94%	-	-
MOV	7.44%	6.84%	0.23%	7.80%
POP_LZC	0.03%	-	-	-
	100%	100%	100%	100%

Table 5.8: Instructions that play a role in determining the Branch Condition Register

INST.	S0		A0	
	Original (%)	Hand-Opt. (%)	Original (%)	Hand-Opt (%)
LD	7.57	5.67	0.01	0.00
FP_MUL	0.00	0.13	0.00	0.00
FP_ADD	6.44	5.20	0.00	0.00
S_ADD	24.89	26.65	0.00	0.00
A_ADD	0.00	0.00	58.96	44.71
A_MUL	0.00	0.00	0.00	0.00
A_SPILL	0.00	0.00	8.40	10.07
S_SPILL	19.51	16.84	0.00	0.00
S_LOGIC	40.18	42.00	0.00	0.00
S_SHIFT	1.41	3.51	0.00	0.00
RECIPR	0.00	0.00	0.00	0.00
MOV	0.00	0.00	32.63	45.22
POP_LZC	0.00	0.00	0.00	0.00
	100%	100%	100%	100%

Table 5.9: Instructions that produce the final Branch Condition

opportunity to improve the branch-stall time here. Scalar adds produce a quarter of the S0 branch conditions, however, and it might be worthwhile to provide fast sign-bit determination in the scalar functional unit to enable earlier issue of branch instructions.

Several other issues affect the execution speed of branches and instruction overlap. We examine some of them in the next section.

5.3. BRANCH EXECUTION

In this section we study some factors specific to the Cray machines that affect branch execution speed. First, we examine the distances of the branch targets from the branch instructions. Given the presence of instruction buffers on the CRAY Y-MP, the distance of the branch-target instruction from the branch instruction determines the likelihood of a cache-hit for the target instruction if a branch is taken. (We note that the penalty for a cache-miss on the CRAY Y-MP is high because of the long memory latency.) These distances determine the range of instruction buffer sizes that would be most effective. Instructions on the CRAY Y-MP can be 1, 2, or 3 parcels (16 bits per parcel) long, and the instruction cache is 512 parcels in size. Furthermore, the instruction cache is partitioned into four buffers of 128 parcels each; when the next instruction to be fetched is in a different buffer than the current instruction, a 2 clock-cycle delay is incurred in fetching the next instruction.

Tables 5.10, 5.11, 5.12, and 5.13 show the branch-target distances for various kinds of branches. The targets of subroutine calls are, as expected, very far from the

Target (parcels)	compiler-opt.			hand-opt.	
	Scalar	Moderate	Vector	Moderate	Vector
1-25	0.0	0.3	0.0	0.0	0.0
26-63	0.0	0.3	0.4	0.0	0.3
64-127	0.0	0.3	0.4	0.2	0.6
128-255	0.5	0.5	0.9	0.5	0.6
256-511	0.6	0.6	1.0	3.7	0.9
512-1023	3.6	2.4	1.5	4.7	1.3
1024-2047	5.9	4.5	2.7	7.4	2.4
2048-4095	6.2	5.4	4.2	13.1	3.7
>=4096	100.0	100.0	100.00	100.0	100.0

Table 5.10: Subroutine-Call Branches

calling instructions. However, although each instruction buffer has consecutive memory locations stored in it, different buffers can contain instructions from different parts of memory (i.e., each buffer can be thought of as a cache-block). Thus, if a subroutine is called frequently in a tight loop, it could exist in one of the buffers while the calling loop can exist in a different buffer, thus preventing a cache miss on each subroutine call/return.

A very large fraction of the targets of unconditional branches (table 5.11) are within the cache size of 512 parcels, and a significant fraction is within half the cache size. This provides a good chance of the target being available in one of the buffers, given that spatial locality is exploited. Instruction-prefetching is an obvious solution for cache-misses for such branches. We note that unconditional branches form less than 10% of all branches (tables 4.16 and 4.17), and hence are not very critical to

Target (parcels)	compiler-opt.			hand-opt.	
	Scalar	Moderate	Vector	Moderate	Vector
1	0.0	0.0	0.0	0.0	0.0
2	0.7	0.0	0.0	0.0	0.0
3-4	1.1	0.0	0.0	0.1	0.0
5-7	1.1	0.0	1.6	8.3	0.0
8-10	18.3	25.3	5.0	8.8	7.4
11-15	19.3	25.3	5.4	10.3	7.5
16-25	35.1	25.5	25.5	10.9	15.3
26-63	48.8	51.2	38.3	37.3	26.8
64-127	54.8	76.0	48.5	40.6	41.6
128-255	68.9	83.5	59.5	47.0	51.4
256-511	75.5	97.9	85.1	75.1	69.2
512-1023	95.5	99.7	99.8	90.2	99.6
1024-2047	98.8	99.7	99.9	100.0	99.9
2048-4095	99.2	99.7	100.0	100.0	100.0
>=4096	100.0	100.0	100.0	100.0	100.0

Table 5.11: Unconditional Branches

Target (parcels)	compiler-opt.			hand-opt.	
	Scalar	Moderate	Vector	Moderate	Vector
1	0.0	0.0	0.0	0.0	0.0
2	0.7	0.0	0.0	9.8	9.8
3-4	1.1	0.0	0.0	12.5	11.8
5-7	1.1	0.0	1.6	13.7	11.8
8-10	18.3	25.3	5.0	13.8	12.4
11-15	19.3	25.3	5.4	13.9	12.4
16-25	35.1	25.5	25.5	16.5	12.7
26-63	48.8	51.2	38.3	48.4	43.2
64-127	54.8	76.0	48.5	80.0	72.9
128-255	68.9	83.5	59.5	86.3	79.4
256-511	75.5	97.9	85.1	94.8	85.5
512-1023	95.5	99.7	99.8	99.5	87.3
1024-2047	98.8	99.7	99.9	100.0	94.1
2048-4095	99.2	99.7	100.0	100.0	94.1
>=4096	100.0	100.0	100.0	100.0	100.0

Table 5.12: Branches based on Register A0

Target (parcels)	compiler-opt.			hand-opt.	
	Scalar	Moderate	Vector	Moderate	Vector
1	0.3	0.2	0.2	1.3	0.0
2	2.2	0.6	0.6	2.6	0.3
3-4	14.8	1.5	1.3	5.7	0.7
5-7	17.4	1.9	1.5	6.4	1.3
8-10	19.3	3.2	1.6	7.7	1.9
11-15	23.7	9.8	5.5	9.7	5.6
16-25	33.2	12.3	7.1	21.2	6.2
26-63	50.6	55.6	46.8	52.1	50.4
64-127	56.8	69.9	76.1	71.0	72.0
128-255	67.8	78.7	81.5	83.5	84.8
256-511	97.4	83.4	86.7	90.6	88.0
512-1023	99.5	98.2	91.7	96.6	91.5
1024-2047	99.6	99.9	94.5	99.8	94.4
2048-4095	99.6	100.0	100.0	99.8	100.0
>=4096	100.0	100.0	100.0	100.0	100.0

Table 5.13: Branches based on Register S0

performance. The CRAY machines do not prefetch targets of unconditional branches.

From table 5.12, we observe that most of the targets of loop-control branches are less than 512 parcels (the cache size) away. This is due to the fact that the compiler limits loop unrolling so that the unrolled loops fit in the cache. This prevents cache misses on each loop iteration which would be extremely detrimental to performance on a long-memory-latency machine like the CRAY Y-MP.

For the data-dependent branches (table 5.13), we again observe that a majority of them are within the instruction cache size. We notice several short branches, especially in the scalar programs. In fact around 70% of the targets have the potential of being found in the same instruction buffer (each of which is 256 bytes long).

Having examined branch target distances, we look at another aspect of branch execution on the Cray machines. On the Cray Y-MP, a branch instruction cannot be issued until 5 clock cycles after the corresponding S0/A0 register has been written. This is an implementation necessity, and could become a detriment to performance unless sufficient number of independent instructions are found to fill the time between when the branch condition is determined (i.e., when S0/A0 is written) and the branch is issued. Table 5.14 shows the number of instructions scheduled between the instruction that produces the branch condition and the branch instruction. We see that although the average is not always equal to or greater than 5 instructions, the compiler is usually successful in filling atleast 3 of the 5 stall cycles, except for the data-dependent branches of the unoptimized codes. Also, usually the distance

Program	S0		A0	
	Original Benchmarks	Hand-Opt. Benchmarks	Original Benchmarks	Hand-Opt. Benchmarks
ADM	6.97	5.89	9.76	8.22
ARC3D	2.96	3.12	3.84	5.08
BDNA	1.74	6.42	9.15	6.58
DYFESM	3.65	3.84	3.78	4.50
FLO52	2.58	3.16	3.67	9.35
MDG	1.74	5.83	3.57	3.84
MG3D	2.72	4.26	2.04	3.85
OCEAN	3.27	3.06	8.85	3.17
QCD	2.28	2.23	7.42	4.12
SPEC77	3.28	3.54	6.15	9.80
SPICE	2.66	2.82	3.31	5.99
TRACK	1.38	5.10	5.52	6.15
TRFD	2.32	3.16	4.00	6.03

Table 5.14: Distance (insts.) between Condition Register update and corresponding branch

between the the update of A0 and the corresponding branch is larger than the distance between the update of S0 and the corresponding branch. This is because most A0-based branches are loop-control branches as discussed before, with only a loop-counter increment involved in updating A0. Thus all the computation of the basic block can be interspersed between the counter update and the branch. S0-based branches are data-dependent branches, and hence many more instructions of the block are involved in determining S0, thus leaving fewer instructions to fill the gap between the update of S0 and the branch.

Given that branches are frequent in scalar blocks, the unfilled slots between the branch condition determination and the issue of the branch instruction could be quite costly. In addition, each branch itself has a minimum execution time on the CRAY Y-MP of 2 clock cycles in the best case, which is a not-taken branch with the fall-through instruction in the same instruction buffer. For other cases where the branch is taken, or the fall-through instruction is not in the same instruction buffer as the branch, or the target instruction is not in the instruction buffers at all, there can be from 4 to 18 cycles of branch execution time. The current CRAY implementations stall instruction issue during these clock cycles. Thus, in the CRAY Y-MP, all instructions of a basic block are issued in program order before the final branch instruction is issued, and after the branch is issued no further instructions can be issued until branch execution is complete. Consequently, although previously-issued instructions can be in execution during the execution of a branch, branches can still prove to be quite costly. Thus, reducing branch-delay is important to improving the speed of scalar-code on the CRAY machines. One scenario where this branch execution cost can be reduced is if a dynamic window of instructions with out-of-order issue from the window is used. In this scenario, if the window is filled with instructions from a basic block, the instructions that determine the branch and subsequently the branch itself could be issued much before the rest of the instructions of the block are issued. Even if the branch takes more than the minimum of 2 clock cycles for execution, there could be instructions from the branch’s basic block that can still be issued, thus filling the "branch-delay slot" that was created at runtime due to slower branch execution. Needless to say, however, such a dynamic dataflow window is not a straightforward extension to the Cray architecture, given the issues of dynamic dependence resolution, maintenance of precise interrupts, etc., that need to be addressed. Note that this "branch-delay slot" is created at runtime, and the function of the dynamic instruction window cannot be moved to the compiler. For the compiler to fill "branch-delay slots", the delay would have to be architecturally specified. Such architectural specification of the branch delay slot does not address the problem discussed above of dynamic variation in branch execution latency.

5.4. FUNCTIONAL-UNIT PIPELINE PARALLELISM

Scalar code is dominated by data- and control-dependencies, and hence shorter latencies rather than larger bandwidths are essential to its fast execution, given some minimum amount of bandwidth. Shorter latencies enable faster completion of the individual instructions and hence earlier issue and execution of all the dependents of an instruction. Large bandwidths will remain unused since dependencies prevent instructions from being issued.

As discussed in chapter 3, the scalar functional units of the CRAY Y-MP are deeply pipelined. The specific latencies of the CRAY Y-MP functional units are, as of date, considered Cray Research Inc. proprietary information. However, the functional units of the CRAY Y-MP have similar pipelines to the CRAY X-MP, except that they are somewhat deeper. Table 5.15 lists the latencies of the individual functional units of the CRAY X-MP.

Kunkel and Smith [Kunkel86] discuss the overheads associated with pipelining a functional unit; as the number of pipeline stages increases, the overall latency of the functional unit increases (in terms of wall-clock time, rather than clock periods). For

Inst. Type	Latency (clocks)
FP_ADD	6
FP_MUL	7
S_ADD	3
A_ADD	2
A_MUL	4
S_SHIFT	3
RECIPR	14
POP_LZC	4
LOAD	14

Table 5.15: Functional-Unit Latencies for the CRAY X-MP

example, if the number of pipeline stages in a functional unit is doubled, the functional unit can only be run at a clock rate that is a little *less* than twice the original clock rate, due to factors such as clock skew and data skew. Similarly, if the clock rate is doubled, the number of pipeline stages may more than double with respect to the original number, not only due to clock skew but also due to the fact that partitioning the combinational logic involved into exactly twice the number of pipeline stages may not be possible. (Note that, conversely, if the clock rate is halved, the number of pipeline stages needed in the functional unit may sometimes be one *more* than half of the original number.) On the whole, shorter pipelines provide shorter real-time functional unit latencies, which is desirable for scalar code.

In this section we examine the utilization of the deep pipelining of the CRAY Y-MP by the scalar code in our benchmarks, to determine whether the current pipelining levels are appropriate. We use the *interarrival times* of instructions to each of the functional units as a measure of the utilization of the pipelining of the unit. The pipelining of a given functional-unit is being fully utilized when that pipeline sees new instructions arriving every clock cycle. In this case, 100% of the interarrival times for the functional unit are 1 clock cycle in duration. A vector instruction achieves such functional unit utilization of its functional unit. In the scalar portion of the Cray machines, such utilization of the functional units is clearly not possible, since on each clock only one instruction is issued while any of the numerous functional units can accept an instruction. Therefore, what we are interested in studying is whether, during certain phases of program execution, the deep pipelining of the functional units is necessary. This would be the case if there exist, in certain stages of program execution, a number of instructions of the *same type* that can be executed in parallel and hence are issued in consecutive clock cycles. The distribution of operations in such parallel phases of the programs plays an important role in determining the utilization of pipelining. If such parallelism is across instruction types, then consecutive instructions will be issued to different functional units, and hence we will see few back-to-back arrivals for any given functional unit type. Back-to-back arrivals imply data-parallelism across the instructions of a particular instruction type. Such data parallelism is not expected to be of a high degree in the scalar portions of a vectorized program, since code segments with such parallelism are expected to be vectorized. Of course, if the compiler fails to vectorize such portions either due to its current limitations or due to the necessity of dynamic information for such vectorization, we will see back-to-back arrivals of instructions to a particular instruction type.

Since we do not simulate program execution, we obtain approximate measurements of the interarrival times as follows. We time each of the scalar basic blocks in isolation (i.e., ignoring the effects of dependencies on other basic blocks as well as memory dependencies, stalls due to memory conflicts, etc.), and measure the interarrival times within each isolated basic block. We use just these measures in our study. Note that we also ignore interarrivals across basic blocks. The error introduced by ignoring the additional stalls due to memory conflicts and similar factors only strengthen our results, since the presence of these factors will only increase the frequency of the *longer* interarrivals while decreasing that of the shorter ones. Now, the error introduced by ignoring the effects of dependencies of the instructions in a basic block on those in other basic blocks also strengthen our results, as explained below. For example, if inter-block arrivals were to be considered, one would have to consider the branch at the end of each basic block. The fastest a branch can execute on the CRAY Y-MP/X-MP is 2 clock cycles, which implies that all interarrival times that span basic blocks will be at least 3 clock cycles. Furthermore, for every two basic blocks that are adjacent to each other in the dynamic execution trace, only *one* instruction type can see this 3-clock interarrival across the blocks, since at most one instruction is issued per clock. All the other instruction types will see interarrivals that are even longer. Since scientific code is dominated by loops, we expect such interarrival periods that are spread across loop iterations to in fact be common and high in number. Furthermore, if data dependencies exist across the two basic blocks (or loop iterations), the interarrival times will dilate further. Thus, considering interarrivals across basic blocks will only *decrease* the fraction of the shorter interarrivals in our estimates.

Consider, for example, a 10 cycle loop, including the branch, that issues an S_ADD instruction on clocks 3, 6 and 7, as shown in figure 5.1. Without counting cross-block arrivals, the interarrival period distributions are: 1 clock (between instructions 5 and 6) - 50%, 3 clocks (between instructions 3 and 5) - 50%. However, considering cross-block arrivals, the fractions are: 1 clock (between instructions 5 and 6) - 33.3%, 3 clocks (between instructions 3 and 5) - 33.3%, 6 clocks (between instruction 6 of one loop-iteration and instruction 3 of the next iteration) - 33.3%. Hence, the conclusions of our study will be made stronger by eliminating the approximations in our measurements.

Tables 5.16 and 5.17 present the interarrival times seen in the scalar codes of the two versions of the benchmarks. First, column 2 ("Isolated" column) presents, for each instruction type, the fraction of all instructions of the type that is the only instruction of that type in its scalar basic block. For example, for the compiler-optimized codes, 29.69% of all the FP_MUL instructions that occur in scalar basic blocks are the only FP_MUL instructions in their basic blocks. These instructions can only contribute to interarrival times that span basic blocks, and such interarrival times are 3 clock cycles or longer, as discussed above. Hence these instructions do not exploit the current level of pipelining of the FP_MUL functional unit. Next, columns 3 through 9 of the tables present the interarrival times for the *rest* of the instructions (i.e., the non-isolated). Columns 3 through 9 of each row thus sum to 100%.

We observe that, *even among the non-isolated instructions*, usually only a small fraction arrive back-to-back (i.e., have 1 clock-cycle interarrival time). For the floating-

CLOCK	INST.	INST.-TYPE
1	1	some_other_inst_type
2	2	some_other_inst_type
3	3	S_ADD
4		/* stall issue */
5	4	some_other_inst_type
6	5	S_ADD
7	6	S_ADD
8		/* stall issue */
9	7	BRANCH /* loops back to inst.1; two-cycle execution */
10		/* wait for branch to complete execution */

Figure 5.1: Example: Measuring Interarrival Times

Functional Unit	Isolated (%)	Inter-Arrival Times (%)						
		CPs						
		1	2	3	4	5	6	>=7
FP_MUL	29.7	7.3	8.3	9.0	5.1	7.9	3.7	58.8
FP_ADD	32.2	9.5	9.5	2.3	5.7	4.3	2.9	65.8
S_ADD	43.2	30.1	11.6	20.6	6.2	5.8	7.9	17.8
A_ADD	36.9	50.9	12.0	9.2	7.8	2.7	1.5	16.0
A_MUL	85.2	18.1	1.1	30.6	2.6	0.3	5.3	42.0
S_LOGIC	48.2	40.0	14.1	7.2	2.5	2.2	1.6	32.5
S_SHIFT	53.9	18.4	9.3	1.4	6.6	17.1	3.6	43.6
RECIPR	85.7	9.3	13.8	0.0	0.0	0.1	0.0	76.8
POP_LZC	81.2	20.0	8.3	3.4	0.0	0.0	0.7	67.5

Table 5.16: InterArrival Times for the Scalar Basic Blocks

— *Compiler-Optimized Codes*

Functional	Isolated	Inter-Arrival Times (%)						
Unit	(%)	CPs						
		1	2	3	4	5	6	>=7
FP_MUL	39.2	13.1	10.4	6.4	4.8	2.7	1.4	61.2
FP_ADD	35.5	8.5	9.9	3.7	3.5	2.6	0.3	71.5
S_ADD	52.2	30.9	10.6	18.4	9.8	10.0	3.4	17.0
A_ADD	38.9	30.9	12.9	13.8	16.2	8.2	5.7	12.3
A_MUL	84.9	55.2	9.3	3.8	0.0	2.3	0.0	29.5
S_LOGIC	51.5	33.7	17.1	25.8	3.1	2.8	0.8	16.8
S_SHIFT	64.5	18.0	24.9	7.9	9.8	8.8	4.1	26.6
RECIPR	80.2	4.8	7.7	0.6	5.6	2.1	2.5	76.6
POP_LZC	90.4	18.6	12.3	2.8	1.1	5.1	0.3	59.8

Table 5.17: InterArrival Times for the Scalar Basic Blocks

— Hand-Optimized Codes

point instructions, at least 60% of the non-isolated instructions arrive more than 7 clocks, the functional unit latency, apart. Thus, any pipelining of the floating-point functional units is immaterial to the execution of *these* instructions. (We will further discuss the pipelining of the floating-point units shortly.) We note again that while some of the instruction types exhibit between 30% and 50% back-to-back interarrivals, these fractions apply only to the non-isolated instructions. For example, in the hand-optimized codes, while 30% of the interarrival times of S_ADDs are 1 clock cycle, we are not considering the 52% of S_ADD instructions that are isolated within their basic blocks when computing this fraction. Similarly, the 55% back-to-back arrivals of the A_MUL instruction apply only to 15% of the instructions; 85% of the A_MUL instructions are isolated in their basic blocks. Thus the reported proportion of back-to-back arrivals is a large upper bound on the back-to-back arrivals seen by the functional units.

Among the functional units, the pipeline with the largest proportion of back-to-back arrivals, and hence the highest utilized pipeline, is the address add pipeline. Note that for each of the functional units, a large fraction of the interarrivals are larger than the latency of the respective functional units. Now, if a functional unit saw no back-to-back (1 clock) interarrivals, we could halve its pipelining level without affecting performance. Similarly, if the functional unit saw *no* 1-clock and 2-clock interarrivals, the level of pipelining can be reduced to a quarter without affecting performance. From the tables, however, we do see a non-negligible fraction of the interarrivals being back-to-back and a non-negligible fraction being 2 clocks apart. Despite this, cutting down the pipelining level to half or quarter may still actually improve performance due to the resulting reduction in latency. We note that some of these back-to-back arrivals might be in a critical phase of the program where the pipelining is important. If the back-to-back arrivals are not in such critical phases, they can be rescheduled for the shallower pipelines without incurring a cost in program performance. However, we note again that our estimates err in favor of pipelining. Of course, the best way to test the importance of the back-to-back arrivals is to reschedule the code for the reduced functional unit latencies and evaluate the reduction, if any, in program execution speed. We do not carry out such a study here. We focus on pointing towards the desirability of reducing pipelining levels, without trying to estimate the best pipelining levels.

An important issue to be remembered is that even though the individual functional units do not have their pipelines well utilized, such pipelining enables the issue stage to issue an instruction to any functional unit every clock cycle. When reducing the number of pipeline stages, the issue stage could still be run at the current speed if busy-bits are used to identify functional units that cannot currently accept an instruction. If the clock rate of the issue stage is reduced in proportion to the reduction in pipelining level of the functional units, the issue stage could become a bottleneck at a certain stage, thus warranting multiple instruction issue. However, we note that the current issue stage utilization is a maximum of 0.4 instructions/cycle (i.e., an instruction is issued only every 2.5 clocks) even in the scalar programs of both versions of the benchmark set (see tables 4.21 and 4.22). Thus the instruction issue stage is unlikely to become a bottleneck for quite significant reductions in the pipelining level. Finally, we note that the floating-point units in the Cray machines are currently shared by the vector and the scalar instructions. In order to improve scalar code performance, providing separate floating-point functional units that have shallow pipelines is essential. Of course, the vector instructions need dedicated floating-point units that have deep pipelines.

An important reason for the low pipeline utilization and the low instruction issue rates could be the fact that the memory latency is relatively very large on the Cray machines (14 clocks on the CRAY X-MP). When a separate scalar processing unit that executes only scalar code is being designed, a data cache for the scalar data might be a viable option[Smith90]. Providing a data cache could result in tremendous improvements in the *average memory latency*, which in turn might boost instruction issue rates as well as pipeline utilization. To explore this situation, we measured interarrival times for the same benchmarks assuming memory latency to be just 1 clock cycle (tables 5.18 and 5.19), which is extremely short latency for the Cray machines. As expected, the shorter interarrivals have become more frequent with the reduction in memory latency. Surprisingly, however, the increase in such frequency is nominal, and the data lead to the same conclusions as in the case of the long memory latency machine. With respect to the shorter memory latency, however, we caution that if the code were rescheduled by the compiler for this latency the interarrival frequencies might change. Our results are obtained by using code that was scheduled for the original memory latencies only.

Functional Unit	Isolated (%)	Inter-Arrival Times (%)						
		CPs						
		1	2	3	4	5	6	>=7
FP_MUL	29.7	12.0	8.8	8.0	4.4	4.1	4.9	57.9
FP_ADD	32.2	15.8	6.4	3.9	3.1	2.3	2.4	66.2
S_ADD	43.2	30.9	11.3	20.0	6.1	5.9	7.9	17.9
A_ADD	36.9	51.1	11.9	9.1	8.0	2.7	2.3	14.8
A_MUL	85.2	18.1	1.1	30.6	2.6	0.3	5.3	42.0
S_LOGIC	48.2	41.4	14.0	7.5	2.6	2.5	1.7	30.4
S_SHIFT	53.9	18.6	9.8	1.4	5.8	20.4	3.6	40.9
RECIPR	85.7	18.7	4.3	1.0	0.0	0.1	0.5	75.3
POP_LZC	81.2	20.0	8.3	3.4	0.0	0.0	0.7	67.5

Table 5.18: InterArrival Times for the Scalar Basic Blocks

— Compiler-Optimized Codes, 1 cycle LD

Functional Unit	Isolated (%)	Inter-Arrival Times (%)						
		CPs						
		1	2	3	4	5	6	>=7
FP_MUL	39.2	17.6	8.6	3.7	4.4	2.7	1.5	61.5
FP_ADD	35.5	14.2	8.4	5.7	1.6	2.0	0.5	67.6
S_ADD	52.2	32.9	10.2	18.5	10.0	10.2	2.6	15.6
A_ADD	38.9	31.0	13.2	13.9	16.2	8.4	6.1	11.2
A_MUL	84.9	55.2	9.3	14.9	0.0	2.3	0.0	18.3
S_LOGIC	51.5	34.4	17.0	26.1	3.1	2.8	1.0	15.4
S_SHIFT	64.5	19.2	24.2	7.6	13.7	4.9	5.2	25.2
RECIPR	80.2	8.5	2.2	0.6	5.6	1.8	4.6	76.6
POP_LZC	90.4	18.6	12.3	2.8	1.1	5.3	0.0	59.8

Table 5.19: InterArrival Times for the Scalar Basic Blocks

— Hand-Optimized Codes, 1 cycle LD

Thus, we believe a very strong case is made for reducing the pipeline stages in the scalar functional units. The resulting increase in pressure on the instruction issue stage might warrant either an increase in the issue bandwidth, or running the issue stage at a faster clock rate than the functional units. Scalar code can be executed significantly faster by thus reducing the functional-unit latencies.

5.5. SUMMARY

In this chapter we presented a characterization of scalar code to evaluate and aid scalar processor design. In particular, we examined data dependencies in the scalar programs from the point of view of exploiting them to improve instruction latencies. Among other things, we observed that up to 25% of all floating-point operations in our benchmarks have no dependents within the same basic block. Among those that do, however, data dependencies between floating-point add and multiply instructions are frequent. The average number of instructions that consume the result of a floating-point operation ranges between 1.15 and 1.6, thus complicating the combination of individual instructions into compound instructions.

Memory load operations frequently feed instructions within the same basic block; at least two-thirds of all memory loads in the scalar blocks of our benchmarks do so. This indicates that the scalar basic blocks suffer from the long memory latency, since it is hard to find 10 or 15 independent instructions from the same block that can be issued during the latency of the load, even though the blocks in our programs are larger than those in non-scientific programs.

With regard to data-dependent branches, memory loads, floating-point operations, and scalar operations are all usually involved in producing the branch condition. Thus, the condition for the data-dependent branch is not likely to be available early in the execution of the basic block compared to the execution of instructions of the basic block not involved in determining the branch, since the floating-point and the memory operations have long latencies. On the other hand, loop control branches mostly involve just address adds, and hence they can be determined very early in the execution of the basic block if desired. Such early determination of the branch can be used for example to allow early issue of instructions from the successor basic block.

We further studied various aspects of branches to understand factors affecting their execution speed on the CRAY Y-MP. In particular, we examined the distances of branch targets, and the hiding of the latency between condition-setting and the corresponding branch. The distance of the target instruction of a branch from the branch determines the likelihood of the target being found in the cache, given that spatial locality is exploited by the cache. For non-subroutine calls, we find more at least 75% of the branch targets are less than 512 parcels (the cache size) away, and at least 60% are less than half the cache size away from the branch instruction. We note that, for the loop control branches the compiler restricts loop-unrolling to the cache size.

On the Cray Y-MP a conditional branch cannot be issued, due to implementation considerations, until 5 clocks after the condition has been written to the register. We find that on the average 3 of these 5 clocks are utilized to issue independent instructions from the same basic block. However, for the data-dependent branches alone, many programs have only around 2 of these cycles used for instruction issue. The hand-optimized benchmarks have more of the cycles used for instruction issue. Overall, reducing this stall time between the condition update and the branch is desirable.

Finally, we examined the utilization of the current pipeline levels of the scalar functional units, and suggested improving their latencies by reducing the pipelining levels, in order to speed up scalar code. We find that only a small fraction (usually much less than 20%) of the arrivals to any particular functional unit are back-to-back, thus suggesting that decreasing the pipelining of the functional units will not hurt the instruction issue rate and at the same time will improve functional unit latency by eliminating pipelining overheads. Such latency reduction is important to speed up scalar code.

The long memory latency of the Cray machines could be largely responsible for instruction issue stalls and the resultant poor utilization of pipelining. To explore this, we reduced the memory latency to 1 clock cycle to model a data cache, and examined the interarrival times at the various functional units. Although the back-to-back arrivals to individual functional units did increase, the change was marginal and the overall data still suggested the same conclusions as above. Thus, we believe that a scalar processing unit with shallower pipelines than the ones found in the CRAY Y-MP would be beneficial to scalar code.

Chapter 6

SUMMARY AND CONCLUSIONS

We summarize in this chapter our study of the CRAY Y-MP processor and some of the more important conclusions drawn in this dissertation. Due to the number of issues studied and the volume of data presented, not all conclusions of the study are mentioned below.

At the end of this chapter, we discuss potential directions for future work on the topic of this dissertation.

6.1. SUMMARY OF STUDY

We carried out a study of program characteristics and machine behavior of a vector processor, the CRAY Y-MP processor, executing a set of scientific applications, the PERFECT Club benchmark suite. We studied two versions of the programs: a version optimized only by the Cray Research, Inc. production FORTRAN compiler, and a hand-optimized version, optimized by a team of Cray Research programmers, that won the 1990 Gordon-Bell PERFECT Award. We examined various aspects of code behavior for both the program versions. We classified the programs based on vectorization level, and examined the behavior of the individual classes. Furthermore, we also examined the behavior of just the scalar basic blocks in all the benchmarks, in order to aid the design of a scalar processing unit tuned to scalar code. Our study sheds light on long-running scientific programs, both compiler-optimized and hand-optimized, executing on vector machines. A characterization of scientific programs has not been reported to date in the literature. Our study also characterizes scalar code found in vectorized programs; such code, while important to program execution speed, has not been studied to date.

6.1. IMPORTANT RESULTS AND CONCLUSIONS

We list below the results obtained pertaining to several specific architectural/implementation issues studied and the conclusions drawn from them. Each subsection below is devoted to a particular issue studied in the dissertation.

6.1.1. Program Vectorization

Program vectorization levels are the key indicators of program speed and efficient execution on a vector machine. The PERFECT Club benchmarks exhibit a wide range of vectorization, in contrast to the usually expected high level of vectorization of scientific programs. The fraction of operations of the benchmarks vectorized when vectorization is carried out by the state-of-the-art Cray Research, Inc. FORTRAN compiler ranges from as low as 4% to as high as 96%. Upon hand-optimization of the benchmarks, the vectorization level improves considerably for many programs, and the percentage of operations vectorized now ranges from as low as 55% to as high as 97%. The hand-optimized version of the benchmarks studied is the version that won the 1990 Gordon Bell-PERFECT award for the fastest supercomputer applications, and was optimized by a team of Cray Research, Inc. programmers. Two of the benchmark programs, SPICE and QCD, show 70% additional vectorization thanks to hand optimizations.

An important conclusion is that the performance attainable using current state-of-the-art vectorizing compilers is much less than the performance available via manual optimizations. Knowledge of the underlying vector machine on the part of the programmer is still very necessary to harness the full power of the machine.

6.1.2. Instruction and Operation Counts

Dynamic instruction and operation frequencies drive the choice of and the organization of the functional units and of the memory ports in the processor. When measuring these frequencies, we classify our benchmarks into vector, moderately-vector, and scalar classes, based on the proportion of vector operations in the programs. Importantly, we observe that the proportions of instructions and operations of each class of programs is not affected by whether the programs were compiler-optimized or hand-optimized. Thus, for example, the moderately-vectorized programs of both the compiler-optimized and the hand-optimized versions have generally similar characteristics. Thus, for our benchmarks and for the hand-optimizations carried out on them, instruction and operation counts are determined by the vectorization level and not by how the vectorization level was achieved. Consequently, machine design vis-a-vis operation frequency is not affected by the presence of hand-optimization beyond the fact that the hand-optimized codes contain no programs in the scalar class.

The vector programs are predominantly floating-point operations and memory accesses, as expected. The moderately-vector and the scalar programs, with their lower vectorization levels, have significant proportions of integer as well as address-computation operations (which, while being integer calculations, are distinguished from other integer operations) in addition to floating-point and memory operations. Thus adequate attention needs to be paid to these instruction types in processor design. Very interestingly, on the CRAY Y-MP, spill operations that move values between the scalar primary register sets and the corresponding backup register sets and move operations that move values between the two scalar primary register sets together constitute the largest fraction of operations in scalar programs as well as in the non-vectorized basic blocks of the moderately-vector and vector programs. For example, these miscellaneous operations, which are specific to the Cray machines, constitute close to 40% of all operations of the scalar programs of the compiler-optimized benchmarks. While the spill instructions can be eliminated by using a larger register set, the impact on clock cycle, instruction format, etc., have to be considered when increasing the register set size. Furthermore, these miscellaneous operations are single-cycle operations whereas most other operations are multi-cycle operations. Hence there is an opportunity for most of these operations to be executed for free by being overlapped with the execution of the multi-cycle operations.

Almost all the memory operations of the moderately-vector and vector programs are vectorized; furthermore, a very significant fraction of the memory operations of even the scalar programs are vectorized. For example, though only about 14% of all operations of TRACK are vectorized by the compiler, close to 50% of its memory operations are vectorized. Thus, many of the memory references are predictable, and large memory bandwidth is critical to the fast execution of these programs. Latency is consequently a secondary concern for the predictable memory references. We address memory latency issues for scalar blocks in a later section.

Branches form only about 6% of the operations of the scalar programs, and are around 2% of all operations in the moderately-vector programs, and only around 0.3% of all operations in the vector programs. Eliminating the spill instructions from the scalar programs will increase the proportion of branches to about 10%, but the frequency is still much less than the 20% reported for non-scientific programs. In the scalar programs close to half the branches are data-dependent branches, and about a quarter are subroutine calls. In the moderately-vector and the vector programs, the majority of the branches are expectedly loop-control branches.

6.1.3. Basic Blocks

The sizes of basic blocks are important in the choice of compiler techniques for code optimization and scheduling. Basic blocks in non-scientific applications are reported to be around 5 instructions long, and heuristic compiler algorithms tuned to this size are hence used in compilers for general-purpose programs. We find that in our benchmarks small blocks as well as large blocks, including those that are more than 100 instructions in size are frequent. Furthermore, the large blocks contain a number of vector instructions and hence contribute significantly to the dynamic operation count and thus to program execution time. Hence, code optimization and scheduling techniques geared toward large blocks are desirable for compilers of scientific code, in addition to techniques geared toward small blocks. The Cray Research, Inc. compiler already incorporates several such algorithms.

Apart from the vectorized blocks, the blocks in the scalar programs as well as the scalar blocks in the non-scalar programs have median block sizes between 8 and 10 instructions, but are widely distributed in the range of 1 to 20 instructions and beyond. In fact, for the compiler-optimized benchmarks, 10% of the scalar basic blocks of all program classes are at least 20 instructions in size. Thus, large basic blocks are seen even in scalar codes.

6.1.4. Instruction and Operation Issue Rates

Parallel instruction issue is the topic of much current research in the area of uniprocessor performance. The instruction issue rates on the CRAY Y-MP lie between 0.1 instructions per cycle and 0.45 instructions per cycle for both the hand-optimized and the compiler-optimized versions of the programs. This suggests that instruction issue is not a bottleneck in the CRAY Y-MP, on the average, for current compiler capabilities of exploiting instruction level parallelism. Since each vector instruction issues multiple operations, the operation issue rates are expectedly higher, ranging between 0.4 and 2 operations per cycle for the compiler-optimized benchmarks and between 1.0 and 2.5 operations per cycle for the hand-optimized benchmarks. The deeply pipelined functional units and the long memory latency result in much of the operation-level parallelism being exploited by pipelining and hence we see comparatively less parallelism at the issue stage. Furthermore, the quantity of vector resources in the Cray machines limits the exploitation of the parallelism between vector instructions, as has been reported in the literature.

6.1.5. Data Dependencies in Scalar Code

We studied data dependencies in the scalar portions of the programs to investigate the potential for using compound functional units to speed up the execution of scalar code. We restrict our study to dependencies within basic blocks, due to the limitations of our methodology. At the same time, however, it is more difficult to exploit dependencies that cross basic block boundaries. We discussed the difficulties in Chapter 5.

Compound functional units can be used to exploit frequent data dependencies seen in programs. For example, the IBM RS6000 has a compound multiply-add unit to exploit dependencies between floating-point adds and multiplies. Up to 25% of all floating-point operations in our benchmarks have no dependents within the same basic block. Among those that do, however, data dependencies between floating-point add and multiply instructions are frequent. When considering a compound floating-point functional unit to exploit these dependencies, it is important to observe that the average number of instructions that consume the result of a floating-point operation ranges between 1.15 and 1.6, necessitating a smart compiler to handle the combination of instructions into compound instructions. The difficulties involved in the exploitation of instructions that have multiple dependents are discussed in section 5.2.

Memory load operations frequently feed instructions within the same basic block; at least two-thirds of all memory loads in the scalar blocks of our benchmarks do so. This indicates that the scalar basic blocks suffer from the long memory latency, since it is hard to find 10 or 15 independent instructions from the same block that can be issued during the latency of the load, even though the blocks in our programs are larger than those in non-scientific programs.

With regard to data-dependent branches, memory loads, floating-point operations, and scalar operations are all usually involved in producing the branch condition. Thus, the condition for the data-dependent branch is not likely to be available early in the execution of the basic block compared to the execution of instructions of the basic block not involved in determining the branch, since the floating-point and the memory operations have long latencies. On the other hand, loop control branches mostly involve just address adds, and hence they can be determined very early in the execution of the basic block if desired. Such early determination of the branch can be used for example to allow early issue of instructions from the successor basic block.

6.1.6. Branch Execution in Scalar Code

Fast execution of branches is one of the critical factors for speeding up scalar code execution. We studied two important issues related to fast branch execution on the CRAY Y-MP.

The distance of the target instruction of a branch from the branch instruction determines the likelihood of the target being found in the instruction cache, given that spatial locality is exploited by the cache. For non-subroutine calls, we find at least 75% of the branch targets are less than 512 parcels (the cache size) away, and at least 60% are less than half the cache size away from the branch instruction. We note that, for the loop control branches the compiler restricts loop-unrolling so as to fit in the cache.

On the Cray Y-MP a conditional branch cannot be issued, due to implementation considerations, until 5 clocks after the condition has been written to a register. We find that on the average 3 of these 5 clocks are utilized to issue independent instructions from the same basic block. However, for the data-dependent branches alone, many programs have only around 2 of these cycles used for instruction issue. The hand-optimized benchmarks have more of the cycles used for instruction issue. Overall, reducing this stall time between the condition update and the branch is desirable.

6.1.7. Scalar Functional Unit Pipeline Parallelism

We examine the necessity and the effectiveness of the pipelining of the scalar functional units by studying the interarrival times of instructions to each of the functional units. We find that only a small fraction (usually much less than 20%) of the arrivals to any particular functional unit are back-to-back, thus suggesting that decreasing the pipelining of the functional units will not hurt the instruction issue rate and at the same time will improve functional unit latency by eliminating pipelining overheads due to clock skew and data skew. Such latency reduction is important to speed up scalar code.

The long memory latency of the Cray machines could be largely responsible for instruction issue stalls and the resultant poor utilization of pipelining. To explore this, we reduced the memory latency to 1 clock cycle to model a perfect data cache, and examined the interarrival times at the various functional units. Although the back-to-back arrivals to individual functional units did increase, the change was marginal and the overall data still suggested the same conclusions as above. Although this study did not change instruction scheduling to cater to the 1-clock memory latency, we believe that a scalar processing unit with shallower pipelines than the ones found in the CRAY Y-MP would be beneficial to scalar code due to resulting lower latencies.

6.1.8. Conclusions

We studied several issues related to program behavior and machine features in our experimental CRAY Y-MP vector environment. The important contributions of the studies are two-fold in nature:

(1) providing quantitative studies of several hitherto unexplored aspects of vector machines and discussing the impact of these issues on machine design, and

(2) providing quantitative support for several issues that are currently part of "folklore" in computer architecture.

This work sheds light on several different aspects of vector machines, vector programs, the effects of compiler and hand optimizations, and scalar code found in vector systems.

6.2. FUTURE WORK

Architectural changes suggested by the studies reported in this dissertation need to be explored. Most such explorations have to incorporate compiler techniques for exploiting the new architectural features, since the role of the compiler is critical in harnessing the power of the machine.

An extension of the study of vector machines is to consider *execution-time* based behavior, such as the fraction of time instruction issue is stalled waiting for a particular operation type to complete execution. Such studies have not been done in this dissertation due to the limitations of the methodology used. Exact cycle-by-cycle simulations are necessary for such studies, and such simulations are extremely time-intensive when we study benchmarks such as the PERFECT Club programs which execute for hundreds of millions of clock cycles on the CRAY Y-MP.

References

[Adams89]

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.

[Allen87]

R. Allen and K. Kennedy, ''Automatic Translation for FORTRAN Programs to Vector Form,'' *ACM Transactions on Programming Languages and Systems*, vol. 9, October 1987.

[Batche80]

K. E. Batcher, ''Design of a Massively Parallel Processor,'' *IEEE Transactions on Computers*, vol. C-29, 1980.

[Berry89]

M. Berry, et al, ''The PERFECT CLUB Benchmarks: Effective Performance Evaluation of Supercomputers,'' *International Journal of Supercomputing Applications*, vol. 3, May 1989.

[Butler91]

M. Butler, T. -Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, ''Single Instruction Stream Parallelism Is Greater than Two,'' in *The 18th Annual International Symposium on Computer Architecture*, Toronto, Canada, May 1991.

[CDC81]

CDC, ''CDC Cyber 200 Model 205 Computer System Hardware Reference Manual,'' *Control Data Corporation, Arden Hills, MN*, 1981.

[Charle81]

A. E. Charlesworth, ''An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family,'' *Computer*, vol. 14, September 1981.

[Clark88]

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.

[CRI76]

CRI, ''Cray Computer Systems: The CRAY-1 Computing System,'' *Cray Research Inc., Publication No. 2240008B*, 1976.

[CRI84]

CRI, *Cray Computer Systems: CRAY X-MP Model 48 Mainframe Reference Manual*. Mendota Heights, MN: Cray Research, Inc., HR-0097, 1984.

[CRI84a]

CRI, ''The CRAY X-MP Series of Computer Systems,'' *Cray Research Inc., Publication No. MP-2101*, 1984.

[CRI85]

CRI, *Cray Computer Systems: CRAY-2 Hardware Reference Manual*. Mendota Heights, MN: Cray Research, Inc., HR-2000, 1985.

[CRI88]

CRI, ''The CRAY Y-MP Series of Computer Systems,'' *Cray Research Inc., Publication No. CCMP-0301*, February 1988.

[Cybenk90]

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.

[Ellis85]

J. R. Ellis, ''Bulldog: A Compiler for VLIW Architectures,'' Research Report YALE/DCS/RR-364, Department of Computer Science, Yale University, Seattle, WA 98195, February 1985.

[Emer84]

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.

[Eoyang88]

C. Eoyang, R. H. Mendez, and O. M. Lubeck, ''The Birth of the Second Generation: The Hitachi S-820/80,'' *Supercomputing '88*, November 1988.

[Fisher81]

J. A. Fisher, ''Trace Scheduling: A Technique for Global Microcode Compaction,'' *IEEE Transactions on Computers*, vol. C-30, July 1981.

[Fisher83]

J. A. Fisher, ''Very Long Instruction Word Architectures and the ELI-512,'' *Proc. 10th Annual Symposium on Computer Architecture*, June 1983.

[Fisher87]

J. A. Fisher, ''A New Architecture for Supercomputing,'' *Digest of Papers, COMPCON Spring 1987*, February 1987.

[French82]

S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*. Chichester, England: Ellis Horwood, 1982.

[Gajski85]

Daniel D. Gajski and Jih-Kwon Peir, ''Essential Issues in Multiprocessor Systems,'' *Computer*, June 1985.

[Garey79]

M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: Freeman and Company, 1979.

[Goodma85]

J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schecter, and H. C. Young, ''PIPE: a Decoupled Architecture for VLSI,'' *Proc. 12th Annual Symposium on Computer Architecture*, June 1985.

[Gross88]

T. R. Gross, et al, ''Measurement and Evaluation of the MIPS Architecture and Processor,'' *ACM Transactions on Computer Systems*, August 1988.

[Hennes]

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

[Hillis86]

W. Daniel Hillis and Guy L. Steele, Jr., ''Data Parallel Algorithms,'' *CACM*, Dec. 1986.

[Hillis85]

W. D. Hillis, *The Connection Machine*. Cambridge, MA.: MIT Press, 1985.

[HNSX89]

HNSX, ''HNSX Supercomputers Inc.: SX-X Series System Overview ,'' June 1989.

[Hsu85]

W. -C. Hsu, ''Register Allocation and Code Scheduling for Load/Store Architectures,'' Computer Sciences Technical Report #722, University of Wisconsin-Madison, Madison, WI 53706, November 1985.

[Hwu89]

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.

[Jouppi89]

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

[Kohn89]
J. Kohn, ''JUMPTRACE,'' *Cray Research Inc. Report*, April 1989.

[Kunkel86]
S. R. Kunkel and J. E. Smith, ''Optimal Pipelining in Supercomputers,'' *Proc. 13th Annual Symposium on Computer Architecture*, June 1986.

[Lam88]
Monica Lam, ''Software Pipelining: An Effective Scheduling Technique for VLIW Machines,'' *SIGPLAN 1988 Conference on Programming Language Design and Implementation*, June 1988.

[Landsk80]
D. Landskov, S. Davidson, B. Shriver, and P. Mallet, ''Local Microcode Compaction Techniques,'' *Computing Surveys*, 1980.

[McFarl86]
S. McFarling and J. Hennessy, ''Reducing the Cost of Branches,'' in *Proc. 13th Annual Symposium on Computer Architecture*, Tokyo, Japan, June 1986.

[McMaho86]
F. H. McMahon, *The Livermore FORTRAN Kernels: A Computer Test of the Numerical Performance Range*. Research Report: Lawrence Livermore Laboratories, December 1986.

[Mitche88]
C. L. Mitchell and M. J. Flynn, ''A Workbench for Computer Architects,'' *IEEE Design and Test of Computers*, Feburary 1988.

[Miura83]
K. Miura and K. Uchida, ''FACOM Vector Processor System: VP-100/VP-200,'' *Proc. NATO Advanced Research Workshop on High-Speed Computing*, June 1983.

[Nicola84]
A. Nicolau and J. A. Fisher, ''Measuring the Parallelism Available for Very Long Instruction Work Architectures,'' *IEEE Transactions on Computers*, vol. C-33, November 1984.

[Oehler90]
R. R. Oehler and R. D. Groves, ''IBM RISC System/6000 processor architecture,'' *IBM Journal of Research and Development*, vol. 34, January 1990.

[Padua86]
D. A. Padua and M. J. Wolfe, ''Advanced Compiler Optimizations for Supercomputers,'' *CACM*, vol. 29, December 1986.

[Patt85]

Y. N. Patt, W. W. Hwu, and M. Shebanow, ''HPS, A New Microarchitecture: Rationale and Introduction,'' in *Proc. 18th Annual Workshop on Microprogramming*, Pacific Grove, CA, December 1985.

[Pleszk86]

A. R. Pleszkun, G. S. Sohi, B. Z. Kahhaleh, and E. S. Davidson, ''Features of the Structured Memory Access (SMA) Architecture,'' *Digest of Papers, COMPCON Spring 1986*, March 1986.

[Rau88]

B. R. Rau, ''Cydra 5 Directed Dataflow Architecture,'' *Digest of Papers, COMPCON Spring 1988*, February 1988.

[Rubins85]

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

[Russel78]

R. M. Russel, ''The CRAY-1 Computer System,'' *CACM*, vol. 21, January 1978.

[Smith82]

J. E. Smith, ''Decoupled Access/Execute Architectures,'' *Proc. 9th Annual Symposium on Computer Architecture*, April 1982.

[Smith83]

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.

[Smith90]

J. E. Smith, W. -C. Hsu, and C. Hsiung, ''Future General Purpose Supercomputer Architectures,'' *Supercomputing '90*, November 1990.

[Smith89]

M. D. Smith, M. Johnson, and M. A. Horowitz, ''Limits on Multiple Instruction Issue,'' in *Proc. ASPLOS-III*, Boston, MA, April 1989.

[Sohi89]

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

[Sohi90]

G. S. Sohi and W.-C. Hsu, ''The Use of Intermediate Memories for Low-Latency Memory Access in Supercomputer Scalar Units,'' *The Journal of Supercomputing*, 1990.

[Tang88]

J. Tang and E. S. Davidson, ''An Evaluation of Cray-1 and Cray X-MP Performance on Vectorizable Livermore Fortran Kernels,'' *Proc. 1988 International Conference on Supercomputing*, July 1988.

[Thornt70]

J. E. Thornton, *Design of a Computer -- The Control Data 6600*. Scott, Foresman and Co., 1970.

[Trelea82]

P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, ''Data-Driven and Demand-Driven Computer Architecture,'' *ACM Computing Surveys*, vol. 14, No. 1, March 1982.

[Wall91]

D. W. Wall, ''Limits of Instruction-Level Parallelism,'' in *ASPLOS-IV*, Santa Clara, CA, April 1991.

[Watana87]

T. Watanabe, ''Architecture and Performance of NEC Supercomputer SX System,'' *Parallel Computing*, vol. 5, 1987.

INSTRUCTION-LEVEL CHARACTERIZATION OF THE CRAY Y-MP PROCESSOR

by

Sriram Vajapeyam

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN — MADISON
1991

Abstract

Evolutionary computer architecture fundamentally relies on information obtained from empirical characterizations of the nature of programs and of dynamic program usage of machine features. While vector architectures have dominated the supercomputer arena for over two decades and promise to continue to provide superior single processor performance on a large class of scientific and engineering applications, detailed empirical characterizations of these architectures and their workload has not been reported to date in the literature. This dissertation fills this void in the empirical understanding of machines by reporting an instruction-level study of a single processor of the CRAY Y-MP, using as benchmarks the scientific and engineering application programs that comprise the PERFECT Club benchmark suite.

The capability of the compiler is key to harnessing the power of a machine today. Hence we study a version of the benchmarks that is automatically optimized and vectorized by the Cray Research, Inc. production FORTRAN compiler. Furthermore, several optimizations that are easily implementable manually provide significant performance improvements over the best efforts of the compiler today. Therefore we also study a version of the benchmarks, hand-optimized by a team of Cray Research, Inc. programmers, that won the 1990 Gordon Bell — PERFECT award for the fastest version of the PERFECT Club benchmarks on any machine. In both cases, we study only the user routines of the benchmarks.

We observe that the vectorization level of the user routines of the programs varies widely for the compiler-optimized version of the programs, as opposed to being uniformly high. While hand optimizations do improve the vectorization level, several benchmarks still have vectorization levels below 80%. Consequently, the performance of the non-vector features of vector machines are important to program performance. We observe that the scalar code in the programs contain several address calculation operations and Cray-specific miscellaneous operations, in addition to the floating-point operations. Hence adequate attention needs to be paid to these instruction types. Towards exploiting the data dependencies in the scalar code for faster execution, we characterize the dependencies within the scalar basic blocks of the programs. We observe that the utilization of the pipeline parallelism of the functional units by scalar code is low. Thus, any latency improvements that result from lower levels of functional unit pipelining will enhance scalar performance.

For the overall programs, we observe that large basic blocks are significant in number and are important to program performance. Thus compiler optimization techniques geared towards large basic blocks are desirable. The level of vectorization of memory operations is usually very high, thus emphasizing the need for memory bandwidth. We observe that for the CRAY Y-MP hardware and for the techniques currently employed by the Cray Research compiler, the peak instruction issue rate of the CRAY Y-MP is quite adequate, especially because of the presence of vector instructions. In addition to all the issues mentioned above, several related issues are discussed and explored in the dissertation.

Acknowledgements

I would like to thank my Ph.D. adviser, Guri Sohi, for his guidance of this dissertation and his support. He has spent a significant amount of time on this work. The members of my thesis committee, Jim Goodman, Mark Hill, Parmesh Ramanathan, and John Beetem, offered several useful comments on this dissertation; Jim Goodman and Parmesh Ramanathan, the "readers" on my committee, provided several detailed comments. Over the years of my graduate studies, Jim Goodman has always had the time to say an encouraging word; he has encouraged this work in particular, and I would like to thank him for that. I would be remiss if I were to fail to thank Wei Hsu, of Cray Research Inc., who initiated this project when I worked as a summer intern at Cray in 1989; but for him this work would not exist. Wei has been a constant source of knowledge and advice since then: he has pointed out many of the issues involved in this work and influenced the directions taken, explained several aspects of the Cray hardware and the compiler, educated me about architecture research in the real world, and spent numerous hours discussing research in general and providing encouragement whenever needed.

I would like to thank the Computer Sciences Department of UW—Madison for providing me a teaching assistantship at the start of my graduate studies and thus enabling me to study at Madison, far away from home. My adviser, Guri Sohi, has supported me with a research assistantship several semesters. Cray Research supported me as a summer intern in 1989 and 1990, and also supported me at Madison a semester; the internships with the "FAST" group at Cray have contributed significantly to my education. I am deeply indebted to IIT-Madras, to Govinda Dasa College and Vidyadayinee High School in Surathkal, and to the other educational institutions in India that laid the foundations of my education. I am also indebted to many excellent teachers I have had in various places over the years.

The friendship of many people in Madison and elsewhere has been invaluable over the years. I would like them to consider this a personal note of thanks; they are far too numerous to list here. Over the last six years in Madison, several people have eased the adaptation to a new environment, helped recreate "India" in a far off place when direly needed at times, and made life enjoyable. I would like to thank them here: my ex-apartment-mates Balu, Jayant, and Srini, and all the other guys at "Erin Street", for all the camaraderie; Gopal, especially for being an excellent apartment-mate during the last year of my graduate studies; Naren, for tolerating my far too frequent "chai" breaks at his place over the years, for patiently listening to (or probably sleeping through!) numerous monologues of mine, and for being ever helpful; KRS, for the invaluable music, and also Sesh.

Finally, I would like thank my parents, sister and brother, and relatives, for their encouragement and support over the years. Most importantly, this thesis is for my

mother — my first teacher — for her continuing encouragement, guidance, care, and many sacrifices over the years.

Table of Contents

List of Tables

Chapter 1: INTRODUCTION

Chapter 2: ARCHITECTURES FOR EXPLOITING FINE-GRAIN PARALLELISM

Chapter 3: STUDY BACKGROUND AND METHODOLOGY

Chapter 4: CHARACTERIZATION OF VECTOR MACHINE PROGRAMS

Chapter 5: CHARACTERIZATION OF SCALAR BASIC BLOCKS

List of Figures

Chapter 1: INTRODUCTION

Chapter 2: ARCHITECTURES FOR EXPLOITING FINE-GRAIN PARALLELISM

Chapter 3: STUDY BACKGROUND AND METHODOLOGY

Chapter 4: CHARACTERIZATION OF VECTOR MACHINE PROGRAMS

Chapter 5: CHARACTERIZATION OF SCALAR BASIC BLOCKS