1990

Exploiting implicit parallelism in SPARC instruction execution

Todd Michael Austin

Rochester Institute of Technology
School of Computer Science and Technology

Exploiting Implicit Parallelism in SPARC Instruction Execution

by

Todd Michael Austin

A thesis submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

Dr. James Heliotis

Dr. Rayno Niemi

Prof. Michael Lutz

July 1990

1. Title of thesis Exploiting Implicit Parallelism in SPARC Instruction Execution

I Todd Austin hereby **grant permission** to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date 11/21/90

ABSTRACT

One way to increase the performance of a processing unit is to exploit implicit parallelism. Exploiting this parallelism requires a processor to dynamically select instructions in a serial instruction stream which can be executed in parallel. As operations are computed concurrently, an execution speedup will occur. This thesis studies how effectively implicit parallelism could be exploited in the Scalable Processor Architecture (SPARC)[9], a reduced instruction set architecture developed by Sun Microsystems. First an analysis of SPARC instruction traces will determine the optimal speedup that would be realized by a processor with infinite resources. Next, an analytical model of a parallelizing processor will be developed and used to predict the effects of limited resources on optimal speedup. Lastly, a SPARC simulator will be employed to determine the actual speedup of resource limited configurations, and the results will be correlated with the analytical model.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

This research was supported by the Xerox Corporation. They donated more than 500 hours of SPARC CPU time that was used to produce the simulation and analysis results contained in this thesis. Many thanks to the advisor of this thesis, Jim Heliotis, for many useful reviews of this document as well as helpful guidance during the research. Also thanks to the remainder of the thesis committee, Rayno Niemi, for his helpful critiques of this work. And as always, love to Emily and Rice Grain.

CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1 Problem Statement

Computer consumers have always had and always will have an insatiable desire for computational power. Computational power is typically measured in terms of throughput. Computer throughput is defined as the number of instructions processed per unit time. Computer architects employ three techniques to increase this throughput.[1]

1. Reduce the machine cycle time. This is usually accomplished through the use of aggressive implementation technologies, and reduced instruction sets (RISC)[10].

2. Reduce memory access time. This is realized through the use of aggressive implementation technologies and hierarchical memory designs (i.e. caches.)

3. Increase concurrent processing of instructions. This is implemented through exploitation of explicit and implicit parallelism.

Explicit parallelism is apparent to the programmer. The language or operating system will provide constructs for process control and synchronization, and the architecture will provide multiple processors on which to run the program. Implicit parallelism does not require any support by the programmer. Exploiting this parallelism requires a processor to dynamically select instructions in a serial instruction stream which can be executed in parallel.

This thesis studies how effectively implicit parallelism could be exploited in the SPARC architecture.[9] First an analysis of SPARC instruction traces is used to determine the optimal speedup that would be realized by a processor with infinite resources. Next, an analytical model of a parallelizing processor is developed and used to predict the effects of limited resources on optimal speedup. Lastly, a SPARC simulator is employed to determine the actual speedup of resource limited configurations, and the results are correlated with the analytical model.

1.2 Previous Work

A large amount of work has been done in the area of architecture optimization through the exploitation of implicit parallelism. One of the earliest works on the subject was Tomasulo's article in the IBM Technical Journal.[2] In it he describes the internals of the IBM 360/91. This processor was developed with what Tomasulo called virtual functional units. By using multiple virtual functional units, the processor was able to support a number of instructions in various stages of execution. His architecture would not support out of order execution or multiple instruction issues per clock tick.

Shortly thereafter, Keller's paper in Computer Survey formalized the subject of detecting implicit parallelism.[3] He also introduced the Principle of Optimality, which describes the conditions for optimal exploitation of implicit parallelism.

Descriptions of commercially successful products that exploited some implicit parallelization are Thorton and Cray's CDC 6600[4], and Cray's Cray-1.[5] Neither architecture could issue instructions out of order or issue multiple instructions per clock tick.

A more powerful implementation is the dispatch stack method described by Acosta, et al in IEEE Transactions on Computers.[1] This technique allows for out of order execution and multiple instruction issues per clock cycle. This technique will

converge upon Keller's Principle of Optimality if adequate resources are allocated to the processor.

A recent implementation of a processor that supports out of order execution and multiple instruction issues per clock tick is the proprietary processor in IBM's new System/6000 series workstations. An entire issue of the IBM Journal is dedicated to the technology used in its development.[13] Unfortunately, there is no only one of each functional unit (integer ALU, floating point ALU, and compare logic) resulting in much of the implicit parallelism being lost. Section 2.3.3 further investigates speedup with the IBM System/6000 configuration.

The technique used to determine optimal speedup is described in a paper by Sohi, et al.[6] In this paper, they determine the optimal speedup available in Cray–1 like instruction traces.

Sohi also provides implementation solutions to the imprecise interrupt problem.[14] This problem plagues processors which perform out of order instruction issues. When an interrupt occurs, these processors cannot uniquely determine where the interrupt occurred. Section 1.3.4 further details this problem.

1.3 Theoretical and Conceptual Development

This section will introduce the notion of implicit parallelism and show how it can be exploited. The relevant details of the SPARC architecture will also be discussed.

1.3.1 Implicit Parallelism in Instruction Traces

Implicit parallelism in instruction traces is the parallelism that can be exploited between non–dependent instructions in a serial instruction stream. A clever architecture can utilize this hidded parallelism to increase the overall throughput of the system.

For an example of how to exploit the implicit parallelism in an instruction stream see Figure 1.1.

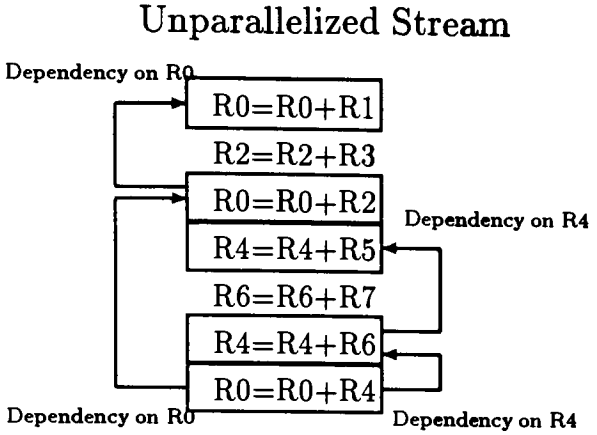


Figure 1.1. Taking Advantage of Implicit Parallelism in Instruction Traces

In the unparallelized stream the dependencies between the different instructions are annotated. In the bottom stream, the instructions have been *parallelized.* This stream was parallelized optimally so now any instruction in the parallelized stream has a dependency with one of the instructions immediately preceding it. The parallelized stream executes in 3 clock cycles whereas the unparallelized stream requires 7 clock cycles. The number of clock cycles needed to execute the code slices can also be termed the *run length.* The side effects from executing either piece of code is identical.

To compare the two streams, a metric of *speedup* and *compression* can be computed. Speedup is defined as

$$S = \frac{R_u}{R_c}$$

where S is speedup, R_u is run length uncompressed, and R_c is run length com-

pressed. In Figure 1.1 the speedup is $\frac{7}{3}$. Compression is a measure of how much the parallelized run length decreased in size and is defined as

$$C = \frac{R_c}{R_u}.$$

For a single run length it is true that

$$C = \frac{1}{S}.$$

Given a series of instructions from an instruction stream, Keller[3] formalized the conditions in which instructions can be executed in parallel. The instructions are assumed to be in the following format.

$$i : OP, S1, S2, D$$

Where i is the instruction label, OP is the instruction operation, $S1$ and $S2$ are the source registers, and D is the destination register. The resulting operation performed by the instruction is as follows:

$$i : D \leftarrow S1\ OP\ S2$$

For each instruction in the stream, a range and domain can be defined. The range is the registers, memory, or condition codes modified by the instruction. That is, the range is the side effect of the instruction. The domain is the registers, memory, or condition codes referenced by the instruction.

If i and j are two different instructions in an instruction stream, then i and j can be executed in parallel if and only if all the following conditions are true.

$$range(i) \cap domain(j) = \emptyset$$

$$range(j) \cap domain(i) = \emptyset$$

$$range(i) \cap range(j) = \emptyset$$

The first condition requires that instruction i's side effects not be referenced by instruction j. If this were not so, a race condition would exist on the memory, registers, and condition codes referenced by instruction j. The second condition asserts the reverse of the first condition. The third condition prevents race conditions that would exist if instructions i and j had the same side effects.

Keller further describes the condition that leads to optimal parallelization.

> **Principle of Optimality**: Whenever j is an operation corresponding to an instruction in the instruction stream, and there is no preceding operation i that is being executed or is pending execution such that i and j cannot be parallelized, then j should be issued.

The resulting parallelized instruction stream may require instructions to execute in a different order than originally found in the code, and may require any number of instructions to be executed in a single clock cycle.

1.3.2 The Effective Scope of Implicit Parallelism

Ideally, optimal speedup would occur if the scope parallelized was the entire execution trace of the program. This would result in a true data flow of the program such that all operations would occur as soon as their operands were available. To accomplish this in hardware would be prohibitively expensive. The hardware would be required to search the entire execution trace for nonrelated activities, and then execute them in parallel.

A more simplistic view of parallelization is usually taken where run length compression only occurs within basic blocks. A basic block is a series of instructions that will not require the parallelizing processor to follow more than one possible stream of execution at a time. The most common delimiter is the conditional

branch. To parallelize past a condition branch would require the processor to execute both streams after the branch, and then drop the results of the unneeded one once the condition code dependency of the branch instruction is resolved. Of course, any number of conditional branches could be encountered, resulting in an exponential explosion in the number of streams being parallelized at once.

The basic block for the SPARC architecture is delimited by the following instructions:

1. conditional branches
2. procedure calls
3. entries into the operating system

The second two delimiters alleviate the need for parallelizing in more than one register set at a time (a characteristic of the SPARC architecture.) Section 1.3.7 further details each case.

1.3.3 The Effects of Compiler Technology on the Extent of Implicit Parallelism

Undoubtedly, the compiler technology used to generate executables affects the amount of implicit parallelism available in the dynamic instruction traces. For example, a compiler that performed register allocation in a low register to high register fashion (i.e. R0, R1; R0, R1, R2, R3; R0, ...) would not provide as much implicit parallelism as a compiler that performed round-robin register allocation (i.e. R0, R1; R2, R3, R3, R4; R6, ...). This is because in the latter case unrelated operations would more likely be in different registers, thus reducing the dependencies between the operations.

In this thesis, speedup will be measured analytically and empirically for the Dhrystone[7] benchmark compiled under: SunOs version 4.1 C compiler, Free

Software Foundation version 1.37.1 ANSI C compiler, TeleSoft version 1.25 Ada compiler, and Xerox version 1.H Cedar/Mesa compiler. All these compilers use different code generators and optimizers, thus it will be possible to see the varying effects on speedup due to compiler technology. The Dhrystone benchmark is a synthetic benchmark. It is crafted to look like a statistically "average" program. Analyzing this program will ensure all frequent operations are included.

1.3.4 Instruction Issue Logic

The instruction issue logic is responsible for deciding when an instruction can be executed. To do this the hardware must determine if the following conditions have been met:

- The instruction issued must not share any dependencies with any other instructions currently executing, or any of the instructions preceding it.
- The appropriate hardware to perform the instructions function must be available.

The logic discussed in this thesis will allow multiple instructions, possibly not sequential, to be issued per clock cycle. Instructions are issued to *functional units.* The functional units perform the desired function (i.e. multiply, add, square root.)

A designer of any instruction issue unit has two degrees of freedom:

- sequential vs. non-sequential instruction issue
- single vs. multiple instruction issue per clock cycle

During each clock cycle, it is the responsibility of the instruction issue logic to dispatch instructions to a functional unit. Once an instruction is issued to a functional unit, a number of clock cycles must pass before the functional unit may

be used again. Often, a number of functional units are available, allowing the processor to support a number of instructions executing in parallel. Pipelining functional units can also increase the number of functional units available.

If the instruction issue capability is limited to only sequential issue, instruction stream parallelization will be limited to neighboring instructions. By issuing instructions out of order, the processor can increase its chances of locating another instruction that is not dependent on the instructions currently executing, or the instructions preceding it. The area of look ahead in which instructions are checked for dependencies is the *instruction window.*

Supporting out of order execution introduces the problem of imprecise interrupts. When an interrupt occurs, the state that needs to be saved is no longer just the program counter, but now is the state (executed, or not yet executed) of every instruction in the instruction window. This problem is further complicated for synchronous interrupts like page faults and divide-by-zero traps. Because a number of instructions are currently under execution, more hardware is required to determine which instruction caused the trap. Restarting a process after interruption is also a complicated procedure. As a result, many high performance architectures do not support virtual memory, and during synchronous traps they only specify the "approximate" location of the offending instruction. Sohi proposes a number of hardware solutions to the problem of the imprecise interrupt.[14]

Another way to increase the amount of parallelization is to allow the processor to issue more than one instruction per clock cycle. This is extremely important for integer instructions, because they typically can execute in one clock cycle. A parallelizing processor that supports multiple instruction issue will look at the current instruction window, and issue all the instructions that do not share dependencies with executing or preceding instructions.

The specific configuration used for simulations in this thesis is the dispatch stack. This instruction issue logic was proposed by Acosta[1]. It supports multiple instruction issues per clock cycle, and nonsequential instruction issue.

1.3.5 The Dispatch Stack

The dispatch stack uses a window of instructions from the head of the instruction stream. As instructions are finished executing, the unfinished, and unissued instructions are pushed to the top of the dispatch stack. The hardware on the dispatch stack is responsible for resolving which instructions can be parallelized, and whether the functional unit resources are available once an instruction is ready to execute. An implementation for this hardware is described in [1]. This issue logic, if given enough resources, will converge upon the issue pattern specified by Keller's Principle of Optimality.

When implementing a dispatch stack, the architect must decide how resources will be spent. The dispatch stack requires two resources to perform its task.

- instruction window slots
- functional units

The number of slots in the dispatch stack (its size) determines how far down the instruction stream the issue logic can look for instructions to issue. The larger the window the better the chance of finding one or more instructions to issue.

The functional unit configuration can also limit the amount of parallelization that can be exploited in the instruction stream. Once the issue logic determines that an instruction can be issued, it must have available the needed functional unit.

1.3.6 The SPARC Architecture

The SPARC architecture was developed by Sun Microsystems.[8] [9] It is loosely based on the RISC[11] and SOAR[12] architectures developed at UC Berkeley. The

processor has a small, simple instruction set that supports both integer and floating point operations in hardware. The processor was first implemented in 1987 by Cypress Semiconductor.

SPARC stands for Scalable Processor Architecture. Because the architecture only specifies the instruction and register set, many implementations are possible. Currently, Cypress Semiconductor, TI, Fujitsu, and BIT have SPARC processor implementations.

Sun Microsystems will sell UNIX kernels to companies that produce SPARC workstations. By using the same UNIX bindings, all the SPARC workstations that use SunOS share binary compatibility for executables. Currently, Sun Microsystems, Solbourne, TI, and BIT sell SPARC based workstations.

1.3.6.1 The SPARC Instruction Set

The SPARC architecture features a simple, orthogonal instruction set. All instructions are 32 bits in length, including any and all operands. This restriction allows for the development of extremely simple and efficient prefetch mechanisms. Table 1.1 lists the SPARC instruction set.

All instructions have exactly one side effect; as a result, no special support is needed for handling page faults. If a memory reference results in a fault, the instruction can be re-executed, no internal state needs to be saved.

Table 1.2 shows the different types of functional units required in the SPARC processor to execute all the instruction types in hardware.

Memory load/store instructions support the following addressing modes.

```
Effective Address = r[rs1] + r[rs2]
Effective address = r[rs1] + sign_ext(simm13)
```

If one of the register sources is not required, the global register $g0$ can be used which always supplies a zero value.

Table 1.1. The SPARC Instruction Set

Opcode	Name
LDSB	Load Signed Byte
LDSH	Load Signed Halfword
LDUB	Load Unsigned Byte
LDUH	Load Unsigned Halfword
LD	Load Word
LDD	Load Doubleword
LDF	Load Floating-Point Register
LDDF	Load Double Floating-Point Register
LDFSR	Load Floating-Point State Register
LDC	Load Coprocessor
LDDC	Load Double Coprocessor
LDCSR	Load Coprocessor State Register
STB	Store Byte
STH	Store Halfword
ST	Store Word
STD	Store Doubleword
STF	Store Floating-Point Register
STDF	Store Double Floating-Point Register
STFSR	Store Floating-Point State Register
STC	Store Coprocessor Register
STDC	Store Double Coprocessor
STCSR	Store Coprocessor State Register
LDSTUB	Atomic Load-Store Unsigned Byte
SWAP	Swap Register with Memory
ADD (ADDcc)	Add (and modify cc)
ADDX (ADDXcc)	Add with Carry (and modify cc)
TADDcc (TADDccTV)	Tagged Add and modify cc (and trap on overflow)
SUB (SUBcc)	Subtract (and modify cc)
SUBX (SUBXcc)	Subtract with Borrow (and modify cc)
TSUBcc (TSUBccTV)	Tagged Subtract and modify cc (and trap on overflow)
MULScc	Multiply Step and modify cc
AND (ANDcc)	And (and modify cc)
ANDN (ANDNcc)	And Not (and modify cc)
OR (ORcc)	Inclusive-Or (and modify cc)
ORN (ORNcc)	Inclusive-Or Not (and modify cc)
XOR (XORcc)	Exclusive-Or (and modify cc)
XNOR (XNORcc)	Exclusive-Or Not (and modify cc)
SLL	Shift Logical Left
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SETHI	Set High 22 bits of Register
SAVE	Save caller's window
RESTORE	Restore caller's window
Bicc	Branch on Integer Condition Codes
FBicc	Branch on Floating-Point Condition Codes
CBicc	Branch on Coprocessor Condition Codes
CALL	Call
JMPL	Jump and Link
Ticc	Trap on Integer Condition Codes
RDY	Read Y Register
WRY	Write Y Register
UNIMPL(n)	Unimplemented Instruction n
IFLUSH	Instruction Cache Flush
FiTO(s,d,x)	Convert Integer to (Single, Double, Extended)
F(s,d,x)TOiR	Convert (Single, Double, Extended) To Integer and Round
F(s,d,x)TOi	Convert (Single, Double, Extended) To Integer
F(s,d,x)TO(s,d,x)	Convert (Single, Double, Extended) To (Single, Double, Extended)
FMOVs	Move Single
FNEGs	Negate Single
FABSs	Absolute Value Single
FSQRT(s,d,x)	Square Root (Single, Double, Extended)
FADD(s,d,x)	Add (Single, Double, Extended)
FSUB(s,d,x)	Subtract (Single, Double, Extended)
FMUL(s,d,x)	Multiply (Single, Double, Extended)
FDIV(s,d,x)	Divide (Single, Double, Extended)
FCMP(s,d,x)	Compare (Single, Double, Extended)
FCMPE(s,d,x)	Compare and Exception if Unsorted (Single, Double, Extended)
CPop	Coprocessor Operate Instructions

Table 1.2. Functional Units Used in the SPARC Processor.

Functional Unit Type	Instructions That Use It
Memory/Cache	Integer/Floating-Point Load and Store
Integer ALU	All other Integer Instructions
FP Mult	FMUL Instruction
FP Divider	FDIV Instruction
FP Square Root	FSQRT Instruction
FP ALU	All other Floating-Point Instructions

All arithmetic instructions use the following three register format.

```
r[dest] = r[rs1] OP r[rs2]
r[dest] = r[rs1] + sign_ext(simm13)
```

To load small immediate values, add $sign\_ext(simm13) + g0$ into the destinations register, where $g0$ will supply a 0 value. To load large immediate values into registers, the SETHI instruction loads a 22 bit value into the highest 22 bits of a register, and zeros the lower 10 bits. If the lower 10 bits need to be set, another add instruction is required. This strategy supports small integer loads in one instruction, while allowing for 32 bit immediate loads with 32 bit length instructions using 2 instructions.

Multiply instructions are only partly supported in hardware. A simple step instruction allows a 32x32 bit multiply to be implemented in 36 clock cycles. Integer division is not supported in hardware. Branch instructions support a 22 bit, sign extended offset from the current PC, and call instructions support a 30 bit sign extended offset from the current PC.

Branch and call instructions each have one delay slot immediately following the execution of the branch or call. The instruction in this slot is executed whether or not the branch is taken. This reduces the number of unfilled slots in pipelined

implementations. The SPARC architecture also supports an "annul" bit for conditional branch instructions. This bit specifies that the instruction in the delay slot will executed as a NOP instruction if the branch is not taken. This option allows compilers to move the first instruction in a loop to the delay slot, thus increasing the chance of finding a useful instruction.

Also included are frame handling instructions and tagged operations. Tagged instructions are intended for use in dynamically typed environments like LISP and Smalltalk. They perform arithmetic operations on 30 bit operands. The 2 least significant bits are reserved for a user defined tag. If the two tags do not match for any operation, an overflow error will occur.

1.3.6.2 The SPARC Register Set

The other interesting feature of the SPARC architecture is the register set. The architecture uses register windows. For any procedure scope the processor provides eight 32 bit global registers, eight 32 bit local registers, eight 32 bit registers for procedure input parameters from the caller (ins), and eight 32 bit registers for parameters to any procedure that the current procedure calls (outs).

The register window works in a circular fashion. When a procedure calls another procedure, the new procedure's ins are the out's of the previous procedure, and the called procedure also receives its own set of eight 32 bit locals. All procedures share access to the same eight 32 bit global registers. Unlike the UC Berkeley RISC implementation, the register window is not mapped into the address space of the processor. Underflow and overflow support gives software the appearance that there are an unlimited number of register windows. The SPARC specification only states that the number of real windows can vary from 2 to 32.

The register $g0$ is a special register for extending the flexibility of many instructions. When referenced it always supplies a 0 value. This is useful when

a one register or immediate only address mode is needed. When written to the information is discarded. This is useful for implementing a compare instruction by doing a subtraction that modifies only the condition code by specifying the destination of the arithmetic operation to be $g0$.

1.3.7 SPARC Characteristics That Limit Run Lengths

Along with the conditional branch, the SPARC architecture also delimits parallelizable run lengths by:

- procedure calls
- entries into the operating system

Procedure calls on the SPARC architecture create a hurdle for issuing instructions. Even though the call instruction is unconditional, it results in the opening of a new register set. If an instruction from the current and next register set are to execute in parallel, the processor hardware must be able to reference both register sets, and realize for dependency checking that they are indeed different. And for significantly large dispatch stacks, any number of register sets could be active at any one time, resulting in references to real registers, and registers that have over/underflowed into memory.

An entry into the operating system is accomplished via the trap instructions and also causes a new register set to be opened. As a result, this operation will also delimit parallelized run lengths.

CHAPTER 2

RESEARCH DESCRIPTION

2.1 Capturing SPARC Instruction Traces

SPARCTrace is a program that has been developed for capturing SPARC processor instruction traces. See Appendix A for user's documentation. It is a general purpose trace tool that creates execution traces with sufficient information such that a simulator could later re-execute the trace. An execution trace is a complete record of all instructions to pass through the execution stage of the processor during the execution of a program. These traces can be very large for some programs. The tool is implemented on SunOS 4.x, thus it will only execute on machines that share binary compatibility with the Sun SPARC workstation series.

Figure 2.1 details operation of the SPARCTrace program. The traced program runs without knowledge that it is being traced. Its StdIn, StdOut, and StdErr file streams remain intact. SPARCTrace uses the Unix PTRACE system call to impose its control over the program being traced. Any SPARCTrace informational output is inserted into the StdErr stream of the program being traced. Instruction stream information is piped to another C-Shell's StdIn stream. A SPARCTrace command line option allows the user to specify the C-Shell command line that will accept the instruction trace stream on its StdIn stream.

2.1.1 Limitations of SPARCTrace

A number of limitations exist which limit the effectiveness of the SPARCTrace program.

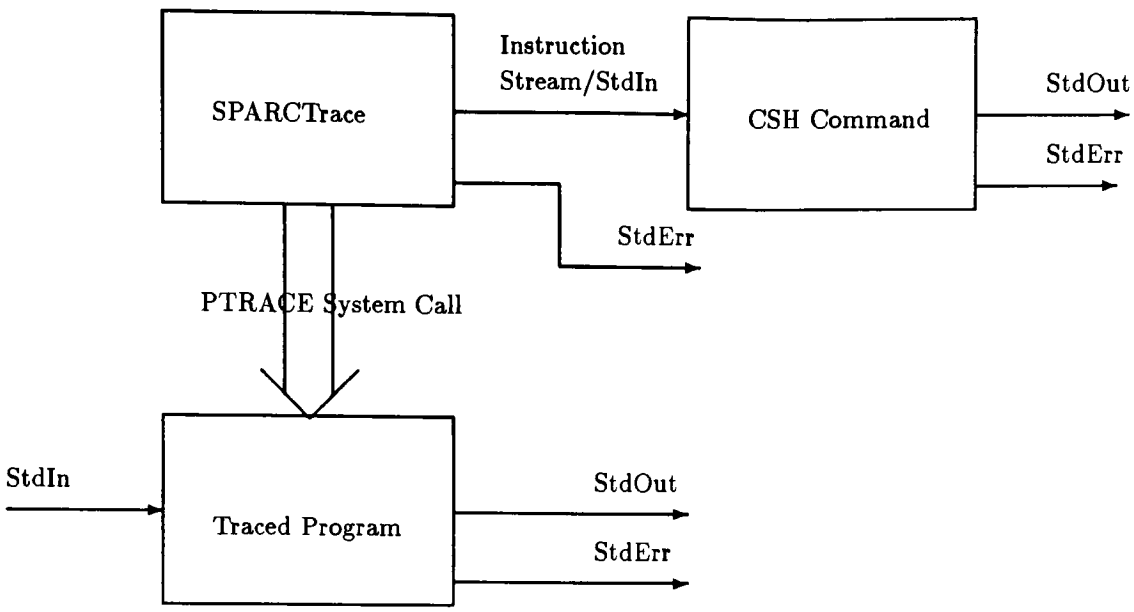


Figure 2.1. SPARCTrace Operation

Because SPARCTrace uses PTRACE to perform program single stepping, it is unable to trace program execution in the operating system. As a result, traces will show entry into the SunOS kernel as a single TRAP instruction. To support the possibility of trace simulation at a later time, the processor state is recorded into the instruction stream after returning from the SunOS call.

A second limitation results from the first. If a program executed an EXEC call to fork off a new process, the SPARCTrace program's output will not include the forked program's execution. This is because the EXEC call is an entry into the OS kernel and SPARCTrace cannot follow this operation.

2.1.2 Instruction Trace Format

As instructions are executed they are placed into the instruction stream. All SPARC instructions are 32 bits, and all operands are contained within the 32 bit instruction.

Three conditions require the insertion of control information into the instruction trace stream.

- Entry into the operating system.
- Execution of a Load/Store Instruction.
- Insertion of debug information into the instruction stream.

The above conditions insert their information into the instruction stream through the use of the UNIMPL instruction opcode. This is an illegal instruction that will not be executed, so it can be used to flag control information in the instruction stream. Table 2.1 details the illegal opcodes used for adding control information into instruction traces.

Table 2.1. Illegal Opcode Formats Used for Inserting Information in Traces

Operation	Opcode	Operands
OS Exit Status	UNIMPL(1)	all registers
Load Integer Single	UNIMPL(2)	dest reg, 32 bit value
Load Integer Double	UNIMPL(3)	dest reg, 64 bit value
Load Floating–Point Single	UNIMPL(4)	dest reg, 32 bit value
Load Floating–Point Double	UNIMPL(5)	dest reg, 64 bit value
Load Floating–Point Extended	UNIMPL(6)	dest reg, 96 bit value
CheckPoint State	UNIMPL(7)	all registers

When a program enters the operating system, it can no longer be traced. If later simulation of the traces is to be performed, the state of the processor must be captured and stored into the trace after exiting the operating system trap. This is required because SPARCTrace cannot track what has changes have occurred to the processor state within the operating system trap. By saving the post trap processor state, a simulation can synchronize with the original trace execution.

Of course the actual side effects that the operating system call may have produced are lost. But for all normal programs, the total side effect of any operating system call is reflected by a change in processor state (i.e. register results), which we captured above, and a change in memory (i.e. disk read results in a buffer), which we capture during load and store operations.

Whenever an integer or floating point Load/Store operation executes the simulation can expect a memory value to follow the operation. This will give any later simulation the same view of memory as in the original trace execution.

The last piece of information inserted into the trace stream is debug information. This information consists of processor state dumps, which will here on be referred to as checkpoints. Any later simulation can verify its operation by comparing its processor state to any checkpoint it encounters. If the two differ, the simulator is broken, and the checkpoint interval can be increased through retracing until the simulator bug can be located.

2.1.3 SPARCTrace Internals

SPARCTrace performs instruction tracing through use of the PTRACE Unix system call. PTRACE allows SPARCTrace to start another process such that its execution is halted until SPARCTrace specifies that it should run.

Unfortunately, the Sun-4 SPARC implementation does not support any single step operations in hardware. As a result, SPARCTrace is required to compute all the possible next addresses and put breakpoint instructions into these locations. Then SPARCTrace will resume execution of the program, it will execute one instruction, then execute a breakpoint instruction which halts the program being traced, and resumes the SPARCTrace program.

After each instruction step, the 32 bit instruction is inserted into the trace stream. The trace stream is a pipe to another CSH session. This session is started

through the use of the POPEN system call. The user specifies on the command line of SPARCTrace what CSH command will be executed.

SPARCTrace make 5 operating system calls for every single instruction traced. This makes the tracing operation very slow since operating system entries are very expensive on the SPARC architecture due to the large number of user registers. To help reduce the number of system calls required for each instruction step, there is a cache between SPARCTrace and the traced programs code segment. This reduces the number of system calls required to read the instruction opcode before it is overwritten with a breakpoint instruction. Table 2.2 shows speedup due to adding the code segment cache. Another operating system entry was removed by calculating the next program counter value whenever possible. This removed the need to read the traced program's registers. Under circumstances where the next program counter value depends on register or memory values (i.e. return from procedure, or call through register) an operating system call is made to determine the next value of the program counter.

Table 2.2. Speedup Gained by the Addition of the Code Segment Cache

Without cache	Instr/Sec: 250
	Instr/Hour: 900,000
	OS Calls/Step: 5
With cache	Instr/Sec: 330
	Instr/Hour: 1,188,000
	OS Calls/Step: 4.02010
	Cache Hit Rate: 97.9897878

2.2 Selecting Programs To Trace

A number of categories were identified for determining which programs to select for tracing.

- Classical benchmarks.
- Commonly used programs.
- Computationally intensive programs (both integer and floating point.)
- Interpreters of other languages.
- Programs developed under different compilers.

Classical benchmarks were selected because they are commonly cited in other papers that study architectural performance enhancements. Commonly used programs will show the most benefit to the user, and computationally intensive programs were selected because these programs will gain the most benefit overall. Lastly, a number of programs under different compilers were selected to determine if different compiler technologies affect the amount of implicit parallelism that can be exploited.

2.2.1 The Test Set

Table 2.3 details the test set selected. It shows the program name, and the category that program is classified under.

The Dhrystone benchmark[7] is a synthetic benchmark designed to perform actions like an "average" program. It will be used to investigate the differences in speedup due to compiler technology. The LINPACK benchmark performs a variety of floating point scalar and vector operations. The C-Prolog interpreter will be running a program exercising the append, member, length, and '+' functions. The nroff text processor will be formatting the "detex" manual page. Gcc, the Free Software Foundation version 1.37.1 ANSI 'C' compiler, will be compiling the Dhrystone benchmark. Grep will be searching for the word 'test' in the TeX sources of this thesis. The Fuzzy Bitmap (FBM) halftone tool will be converting

Table 2.3. The SPARC Instruction Trace Test Set

Dhrystone (cc)	Classical Benchmark, Compiler Specific
Dhrystone (gcc)	Classical Benchmark, Compiler Specific
Dhrystone (MESA)	Classical Benchmark, Compiler Specific
Dhrystone (ADA)	Classical Benchmark, Compiler Specific
LINPACK	Classical Benchmark, Computationally Intensive (Floating Point)
C-Prolog	Interpreter, Computationally Intensive (Integer)
nroff	Interpreter, Computationally Intensive (Integer)
FBM Halftone Image	Computationally Intensive (Integer)
XfRoot	Computationally Intensive (Floating-Point)
Invert	Computationally Intensive (Floating-Point)
gcc	Common Program, Computationally Intensive (Integer)
grep	Common Program, Computationally Intensive (Integer)

a 760x450x8 gray scale image of the Hubble space telescope to a 1000x1000x1 bitmap. XfRoot is an X based program, which produces a fractal pattern for the root window of an X display. Invert is fortran program that determines the inverse of a 10x10 matrix.

All the programs are run such that their execution is deterministic. For the programs that used random number generation, the software was modified so that the same random samples are used for each invocation of the program.

2.3 Analysis of SPARC Instruction Traces

This section details the three steps of research performed in this thesis.

2.3.1 Speedup with Unlimited Resources

The first phase of research determines the extent of parallelism available in the SPARC instruction traces if no resource limits are imposed on the processor execution. This information will provide a benchmark upon which the resource limited configurations can be compared. These characteristics are also used by the analytical model to predict speedup with resource limitations. The resource

unlimited processor will have an infinite number of functional units available at any time, and it will look ahead as far as needed given that the following restrictions are observed. [1]

- The processor will not look down two instruction streams if a conditional branch is encountered. Instead, it will halt the issuing of instructions past the conditional branch until the condition code dependency is resolved.
- The processor will not issue instructions past a CALL instruction until after it has executed.
- The processor will not issue instructions past a TRAP instruction until after it has executed.

A number of other assumptions are made concerning the processor implementation that have an impact on the parallelizing algorithm:

1. All non-floating point instructions take one clock tick to execute.
2. All float point instructions execute in the same number of clock ticks as the Cypress 7C601 SPARC implementation.
3. All memory reference instructions execute in one clock tick.
4. The register file contains enough ports to support the sinking and sourcing of information from all functional units simultaneously.

The first assumption is consistent with current SPARC implementations. The architecture was specifically designed so that all non-floating point instructions could be executed in one clock cycle. This can be seen by the multiply step instruction and the lack of an integer divide instruction. The second assumption

[1] See section 1.3.2 for more discussion and justification on these look ahead restrictions.

admits that one cycle floating point operations are unreasonable. Since floating point hardware is a well studied area and the Cypress 7C601 (used in the Sun-4/110) is a high performance implementation, its floating point instruction cycle times will be used in this study. Table 2.4 shows the number of clock cycles required for each functional unit type. The next assumption supports the first in that all load and store operations, integer or floating point, will complete in one clock cycle. This is consistent with the high hit rates seen on today's cache architecture. It is typical to see 97% and more of all load and store request satisfied in one clock cycle due to hits in the cache. The last assumption is consistent with today's high performance architectures, such as the Cray X-MP[15] and the IBM System/6000 processor[13], and will allow the instruction issue logic to maximize the use of the functional units.

Table 2.4. Functional Unit Computation Times

Functional Unit Type	Computation Time (in clock cycles)
Memory/Cache	1
Integer ALU	1
FP Mult	8
FP Divider	12
FP Square Root	16
FP ALU	4

The following characteristics are extracted from the trace.

- Optimal Speedup for a parallelizing processor with infinite resources.
- Distribution of run lengths before parallelization.
- Distribution of run lengths after parallelization.
- Compression in clock cycles vs. run length in instructions. This metric is used primarily for the analytical model.

- Distribution of functional unit usage per cycle for each type of functional unit.

See Appendix B for the algorithms used to parallelize run lengths and extract trace characteristics.

Table 2.5 details the speedup for each program. Also shown are the total number of instructions executed during the program's execution, and the number of functional units required per cycle. The number of functional units needed per cycle is slightly less than the program speedup because some operations, such as unconditional branches, do not require a functional unit to execute.

Table 2.5. Speedup without Resource Limitations

Program	Total Instructions Executed	Speedup	Number FU required/cycle
Dhrystone (cc)	6141818	2.499196	2.437090
Dhrystone (gcc)	5468643	2.248608	2.149846
Dhrystone (MESA)	8353958	1.882323	1.824304
Dhrystone (ADA)	5033711	1.909649	1.681467
LINPACK	14791977	2.820935	2.782032
C-Prolog	795593	2.330711	2.281481
nroff	7887188	1.764496	1.753184
FBM Image Halftone	34408857	1.803576	1.787067
XfRoot	22626680	2.255020	2.140304
Invert	18229308	3.761269	3.738126
gcc compile	9056182	1.957219	1.931637
grep	2594572	1.769226	1.656092

As shown in Table 2.5, a diverse selection of test software exhibited speedups from 1.76 (nroff) to an impressive 3.76 (Invert). The best speedups are from the two Fortran programs: Invert and LINPACK.

Table 2.6 shows the expected uncompressed run length, R_u, and the expected compressed run length, R_c, for each program traced. Also included is the minimum, maximum, and standard deviation for each expected value.

LINPACK, and Invert have the longest average run length. They also exhibited the greatest speedup. This is most likely due to the increased probability of locating independent instructions in long runs. Inspection of their source reveals a high frequency of complex expressions between control constructs. These complex mathematical expressions typically contain many independent subexpressions which would also allow for more run length compression. Overall, an excellent correlation between expected uncompressed run length and speedup is shown in the table.

Figure 2.2 shows an example curve detailing the run length compression expected for an uncompressed run length. This information is used by analytical model to predict the effects of limited window size on program speedup. The plot shown is for the Dhrystone benchmark compiled with the SunOS 'cc' compiler. Also shown is the number of samples that were used to determine the expected compression for each run length.

Tables 2.7, 2.8, 2.9, 2.10, and 2.11 detail the functional unit requirements for each type of functional unit.

Table 2.7 details the integer ALU requirements for each program. Note that even the floating point intensive programs (Invert and LINPACK) require significant integer ALU resources. This is because control operations found in any program, such and loop counting and booleans, will still be implemented with integer arithmetic.

Most programs do not use floating point.[7] Since this mode of computation is typically less efficient than integer operations, most programs avoid them unless absolutely needed. Table 2.8 shows that for the floating point intensive programs tested, their floating point resource requirements are significantly less than their

Table 2.6. Uncompressed and Compressed Run Lengths

Program	R_u	Maximum Minimum Standard Deviation	R_c	Maximum Minimum Standard Deviation
Dhrystone (cc)	7.108594	33 1 18.591555	2.844353	18 1 3.657467
Dhrystone (gcc)	7.483040	33 1 7.633031	3.327855	19 1 5.833420
Dhrystone (MESA)	5.730862	75 1 21.438793	3.044568	18 1 4.486762
Dhrystone (ADA)	4.675169	80 1 34.308998	2.448181	40 1 9.299739
LINPACK	8.855462	86 1 67.875793	3.102925	50 1 15.467480
C-Prolog	7.505029	67 1 30.918451	3.080993	21 1 2.532271
nroff	5.259784	42 1 14.020875	2.980898	20 1 3.579262
FBM Image Halftone	5.640077	47 1 24.007898	3.127164	21 1 8.669597
XfRoot	8.556650	82 1 141.801743	3.794491	27 1 18.840263
Invert	17.506184	140 1 854.787354	3.883375	82 1 30.280806
gcc compile	5.667139	49 1 12.112222	2.895504	26 1 4.957233
grep	4.600573	33 1 8.204561	2.600331	18 1 1.722443

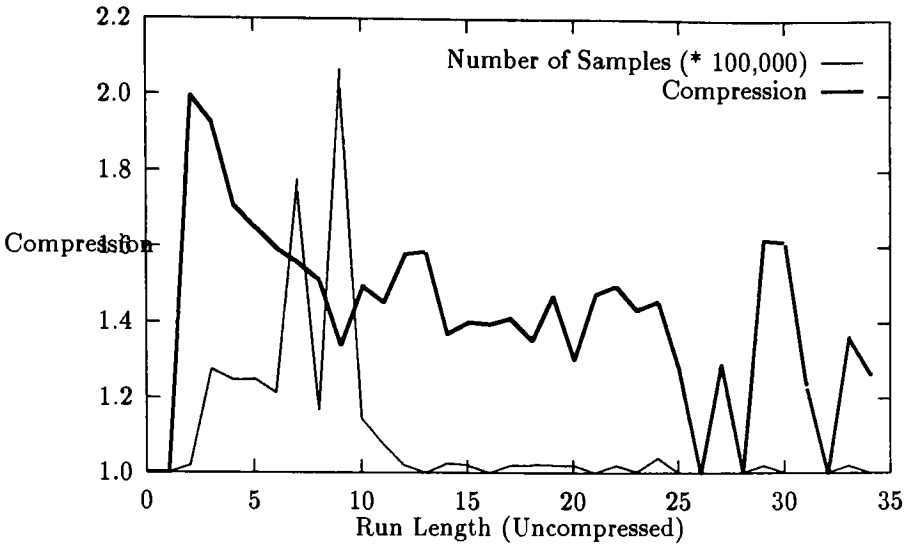


Figure 2.2. Compression by Run Length

Table 2.7. Integer ALU Units Required per Cycle

Program	Int ALU/cycle	Max	Min	Standard Deviation
Dhrystone (cc)	2.437090	13	0	2.154462
Dhrystone (gcc)	2.149846	12	1	2.271023
Dhrystone (MESA)	1.824301	13	0	1.756418
Dhrystone (ADA)	1.681467	18	0	1.020533
LINPACK	1.845362	16	0	1.982322
C-Prolog	2.231284	22	0	3.301412
nroff	1.693120	13	0	1.280882
FBM Image Halftone	1.745964	13	0	1.567700
XfRoot	1.858896	12	0	2.006273
Invert	2.011256	24	0	2.289238
gcc compile	1.885249	16	0	1.670485
grep	1.641571	13	1	0.896194

Table 2.8. Floating Point ALU Units Required per Cycle

Program	FP ALU/cycle	Max	Min	Standard Deviation
Dhrystone (cc)	0	0	0	0
Dhrystone (gcc)	0	0	0	0
Dhrystone (MESA)	0	0	0	0
Dhrystone (ADA)	0.000044	8	0	0.000003
LINPACK	1.287634	14	0	7.998343
C-Prolog	0.000012	4	0	0.000001
nroff	0	0	0	0
FBM Image Halftone	0	0	0	0
XfRoot	0.281408	11	0	1.212014
Invert	1.468640	18	1	8.395570
gcc compile	0.000345	2	0	0.001531
grep	0	0	0	0

integer resource requirements. This implies that a configuration with a resource supply to satisfy most instruction issues will likely have more integer arithmetic logic units than float point units.

Inspection of tables 2.9 and 2.10 show the floating point multiplier and divider resource requirements for all the tested programs. For both resources, the expected number per cycle is well below one. LINPACK requires the most of these resources, yet with only one multiplier and divider, a parallelizing processor will fulfill 99.79% of the multiplier resources and 99.99992% percent of the divider resources. For floating point intensive programs, only about one in every one hundred instructions is a floating point divide. Depending on the system workload, the system performance might be increased if the transistors that were to be used for the divider were spent elsewhere on more rewarding hardware, such a larger on chip cache or more arithmetic logic units. The instruction set could remain unchanged by implementing the divide instruction in software.

Table 2.9. Floating Point Multipliers Required per Cycle

Program	FP Mult/cycle	Max	Min	Standard Deviation
Dhrystone (cc)	0	0	0	0
Dhrystone (gcc)	0	0	0	0
Dhrystone (MESA)	0	0	0	0
Dhrystone (ADA)	0	0	0	0
LINPACK	0.382319	12	0	0.893211
C-Prolog	0	0	0	0
nroff	0	0	0	0
FBM Image Halftone	0	0	0	0
XfRoot	0.039903	2	0	0.038312
Invert	0.232107	6	0	0.572837
gcc compile	0	0	0	0
grep	0	0	0	0

Table 2.10. Floating Point Dividers Required per Cycle

Program	FP Div/cycle	Max	Min	Standard Deviation
Dhrystone (cc)	0	0	0	0
Dhrystone (gcc)	0	0	0	0
Dhrystone (MESA)	0	0	0	0
Dhrystone (ADA)	0	0	0	0
LINPACK	0.003214	2	0	0.002313
C-Prolog	0	0	0	0
nroff	0	0	0	0
FBM Image Halftone	0	0	0	0
XfRoot	0.008867	1	0	0.008789
Invert	0.009744	3	0	0.016315
gcc compile	0	0	0	0
grep	0	0	0	0

Table 2.11. Floating Point Square Roots per Cycle

Program	FP Sqrt/cycle	Max	Min	Standard Deviation
Dhrystone (cc)	0	0	0	0
Dhrystone (gcc)	0	0	0	0
Dhrystone (MESA)	0	0	0	0
Dhrystone (ADA)	0	0	0	0
LINPACK	0	0	0	0
C-Prolog	0	0	0	0
nroff	0	0	0	0
FBM Image Halftone	0	0	0	0
XfRoot	0	0	0	0
Invert	0	0	0	0
gcc compile	0	0	0	0
grep	0	0	0	0

Table 2.11 shows the floating point square root requirements of all the tested programs. Note, none of the test programs traced ever issued the floating point square root instruction. Further investigation of the sources of Invert, XfRoot and LINPACK indicate that they do indeed perform many square root operations. All square root operations on the Sun SPARC workstations result in a call to the math library function _sqrt. This library function implements double precision square roots without the use of the fsqrt instruction. To determine if this was due to a discrepancy between the SPARC specification and Sun's implementation, a simple program was modified such that all calls to the _sqrt function became an invocation of the fsqrt instruction. The program executed properly, so it appears that the SPARC hardware on a Sun 4/110 supports the fsqrt instruction. A message was posted to comp.sys.sun to get more information. A number of Sun engineers responded. It seems original Sun 4/110 hardware did not support the fsqrt instruction, instead a software interrupt is generated whenever the instruction is executed. The interrupt code then calls the floating point square root emulation

routine. The SunOS compilers do not generate the fsqrt instruction, but rather generate calls the fsqrt emulation code directly. This alleviates the overhead of a synchronous software interrupt. The synchronous software interrupt is supported so that the Sun workstations are binary compatible with the Sun SPARC specification. Recently, many Sun SparcStation implementations have had the fsqrt instruction implemented in the processor, but since all Sun SPARC workstations use the same SunOS software the user must link in a special version of _sqrt to take advantage of the fsqrt instruction. Unfortunately, the modified _sqrt function was not received until all analysis and simulation was completed, so no fsqrt results are currently available.

2.3.1.1 The Effects of Compiler Technology

The Dhrystone synthetic benchmark was compiled under four different compilers and traced to determine the effect of compiler technology on implicit parallelism. The maximum level of optimization was used to produce each executable. Table 2.12 shows the optimal speedup available in each programs execution.

Table 2.12. Speedup vs. Compiler Technology

Program	Total Instructions Executed	Total Cycles to Complete	Speedup
Dhrystone (cc)	6141818	2457518	2.499196
Dhrystone (C++)	6021398	2377313	2.532859
Dhrystone (gcc)	5468643	2432013	2.248608
Dhrystone (MESA)	8353958	4438110	1.882323
Dhrystone (ADA)	5033711	2635935	1.909649

The characteristics of the language do not largely affect the results, as the dhrystone benchmark can be expressed identically in all languages. Also, special

language features were not used, such as the more optimized string implementation in Mesa. All operations were performed identically in each language. The resulting comparison should only exhibit effects from the code generation and optimization in the different compilers.

The three 'C' based programs showed very similar speedup, with the 'gcc' compiled program slightly lagging. This also coincides with a slightly more optimized output from the 'gcc' compiler as compared to the output of the Sun 'cc' compiler. The possibility of compiler optimization affecting speedup is discussed in section 2.3.1.2.

An interesting case to note is the Mesa compiler results. The Mesa compiler for the SPARC produces 'C' code suitable for the Sun's cc compiler. But the code is not normal 'C' code but rather a "portable assembly" sources file that uses only a limited number of 'C' constructs. This evidently produces much slower code (more CPU cycles) and supplies less implicit parallelism to the processor.

2.3.1.2 The Effects of Compiler Optimization

This section poses the question: Does the optimization strategies used in the compiler actually reduce the amount of implicit parallelization for the processor? Table 2.13 shows the speedup from code optimization and parallelization. Since the actual speedup of an optimized program includes both the speedup from optimization and from exploiting implicit parallelism, the table also computes the actual speed up from the original, unoptimized version. The benchmark was compiled using the SunOS 'cc' compiler.

In this example, the optimization reduces the amount of implicit parallelism available in the instruction execution. Although is this by no means a complete study of the effects of optimization on speedup, this does demonstrate that optimization and speedup can conflict on a parallelizing processor. Virtually no

Table 2.13. Speedup vs. Compiler Optimization

Program	Speedup from Parallelization	Speedup from Optimization	Actual Speedup
Dhrystone (unoptimized)	2.599854	1.000000	2.599854
Dhrystone (optimized)	2.306735	2.582007	5.956005

published research exists concerning code generation for parallelizing architectures. This is because all of todays truly parallelizing architectures only exist as simulators. When parallelizing processors begin emerging on the market, compiler optimization technology will have to reevaluate the tradeoffs and techniques used so that multiple functional units and limited window sizes are utilized most effectively, while the implicit parallelism speedup and optimization speedup product is maximized.

2.3.2 Analytical Analysis of Speedup with Limited Resources

Next, an analytical model of the parallelizing processor is derived. By applying the trace characteristics from the previous sections, it will be possible to predict the effects of limited resources on execution speedup. Analytical models are important because they allow computer architects to search design space with the application of an equation rather than application of a simulator on captured instruction traces. The resource limited simulator tools used in this thesis can process only about 10 million instructions in a 24 hour period.

Speedup is defined as the number of instruction cycles required to execute a program unparallelized, divided by the number of clock cycles to execute the program parallelized. In equation form:

$$S = \frac{N_{cycles\ uncompressed}}{N_{cycles\ compressed}}.$$

The speedup can also be derived from trace characteristics.

$$S = \frac{R_u}{R_c}$$

where R_u is the expected value of a run length uncompressed[2] , and R_c is the expected value of a run length after being parallelized. The expected value of a run length is

$$R = \sum_{i=0}^{\infty} i \; p_{rl}(i)$$

where R is the expected run length, in clock cycles, and $p_{rl}(i)$ is the probability of a run length being i clock cycles in length. $p_{rl}(i)$ is a characteristic that can be measured from the trace before and after parallelization.

R_u, Uncompressed Run Length

i0, i1, i2, ...

R_c, Optimally Compressed Run Length

i0, i1, i2, ...

R'_c, Expansion Due to Limited Window Size

i0, i1, i2, ...

R''_c, Expansion Due to Limited FU's

i0, i1, i2, ...

Figure 2.3. Derivation of Expected Run Length with Limited Resources

[2]Whenever an expected value is computed it will also be accompanied by its standard deviation. The standard deviation, σ^2, is

$$\sigma^2 = \sum_{i=0}^{i=\infty} (x_i - \mu)^2 p(x_i)$$

where μ is the expected value of x.

Figure 2.3 shows how the resource limited speedup is computed. This is accomplished by determining the expected run length for parallelization using limited resources. Two conditions prevent optimal compression of run lengths.

- The run length to be parallelized is larger than the instruction window size.
- The appropriate functional unit was not available when the issue logic found an instruction to issue.

In this model, the two conditions are assumed to be unrelated. When computing the degree to which each effects expected run length, the other condition will not be part of the equation.

2.3.2.1 The Effects of Limited Instruction Window Size

Figure 2.4 shows how the instruction window size, W, affects the expected run length. For all run lengths less than or equal to W instructions in length, the run will be fully optimized by the hardware, and a new run length of R_c, the optimal expected run length, can be expected. For run lengths longer than W, the entire run will not fit into the instruction window, and a new run length of something larger than R_c can be expected.

The expected run length with limited window size, R'_c, is

$$R'_c = p(R_u \leq W)R_c + p(R_u > W)R_W$$

where R_W is the expected run length for unparallelized run lengths that do not fit into the instruction window.

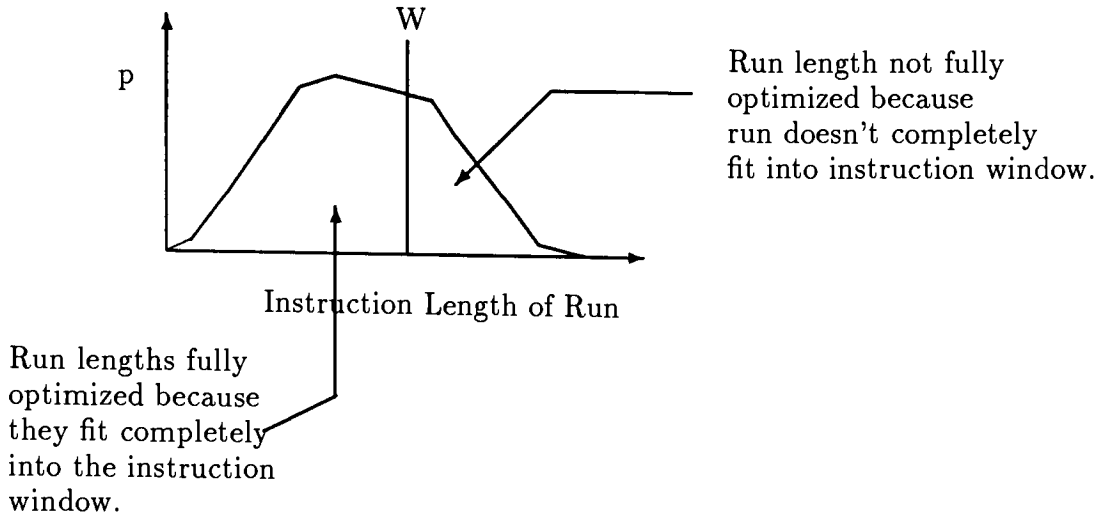


Figure 2.4. Effect of Limited Instruction Window Size on Expected Run Length

The probability that an unparallelized run length is less than or equal to W, $p(R_u \leq W)$, is

$$p(R_u \leq W) = \sum_{i=0}^{W} p_{rl}(i).$$

The probability that the unparallelized run length is greater than W, $p(R_u > W)$, can then be easily computed.

$$p(R_u > W) = 1 - p(R_u \leq W)$$

Next, R_W must be determined. This is the run length expected for all runs longer than W in length. These runs will will not fit completely into the instruction window. As a result, the hardware will not be able to parallelize instructions greater than W slots away. The degree to which this is done in the optimal case will determine the expansion on the optimal run length R_c.

Each clock cycle, the parallelizing processor will "evacuate" $1 + C'$ slots of the instruction window. One slot is emptied due to the processor executing the foremost slot of the instruction window. The C value is the part of the instruction window emptied due to compression in the instruction window. It is assumed that C' in a

window of size W is consistent throughout the trace. As a result, this value can be determined by computing the expected number of cycles needed to execute a run equal to the expected run length for all runs longer than W. In equation form

$$R_W = \frac{R_{u>W}}{1 + (1 - C_{for\ length\ W})\dot{R}_{u>W}} .$$

Where $R_{u>W}$ is the expected run length for all run lengths larger than W instructions in length. It is computed by

$$R_{u>W} = \sum_{i=(W+1)}^{\infty} i\ p_{rl}(i).$$

It could be suggested that the opportunities for parallelization are evenly distributed, thus C can be derived from the expected compression of the average run length for all run larger than W in length. In equation form this is

$$R_W = \frac{R_{u>W}}{1 + \frac{WC_{for\ R_{u>W}}}{R_{u>W}}}.$$

This method turns out to be much to optimistic. For the longer run, $Ru > W$, compression is typically quite high. This method was then dropped in lieu of the previously discussed method.

2.3.2.2 The Effects of Limited Functional Unit Resources

The expected run length will also expand due to instruction cycles in which all the functional units required where not available. The new expected run length is

$$R''_c = R'_c\{p(N_{fu\ needed} \leq N_{fu\ available}) + p(N_{fu\ needed} > N_{fu\ available})Cy_F\}.$$

This equation is detailed in Figure 2.5. The term, R'_c, is the expected run length in clock cycles after expansion from a limited window size is computed. The next term is the average number of clock cycles needed to execute all instructions in an instruction slot. This value is larger than one because not all instructions can be

dispatched due to functional unit resource limitations. The probability that the instruction cycle gets all the functional units required, $p(N_{fu\ needed} \leq N_{fu\ available})$, is

$$p(N_{fu\ needed} \leq N_{fu\ available}) = \sum_{i=0}^{N_{fu\ available}} p_{fu\ needed}(i).$$

The probability that the instruction cycle does not get all the needed functional units can then be easily computed.

$$p(N_{fu\ needed} > N_{fu\ available}) = 1 - p(N_{fu\ needed} \leq N_{fu\ available})$$

Cy_F is the expected cycle length when all the needed functional units are not available. In equations form it is

$$Cy_F = \frac{E(fu\ needed > fu\ available)}{N_{fu\ available}}$$

where $E(fu\ needed > fu\ available)$ is the expected number of functional units required when the number of functional units required is more than available. This value is

$$E(fu\ needed > fu\ available) = \sum_{i=N_{fu\ available}+1}^{\infty} i\ p_{fu\ needed}(i).$$

$p_{fu\ needed}$ is a trace characteristic, measured in the previous section.

The only extension needed to the above model is support for multiple types of functional units. This is accomplished by computing the expansion for each functional unit in succession, where the expected run length to the next computation is the result from the previous.

Using this model, the speedup that can be expected with limited instruction window size, and a limited number functional units is

$$S = \frac{R_u}{R_c''}.$$

This analytical model will be applied in the following section, where its results can be correlated to the actual empirical speedups measured with the resource limited simulator.

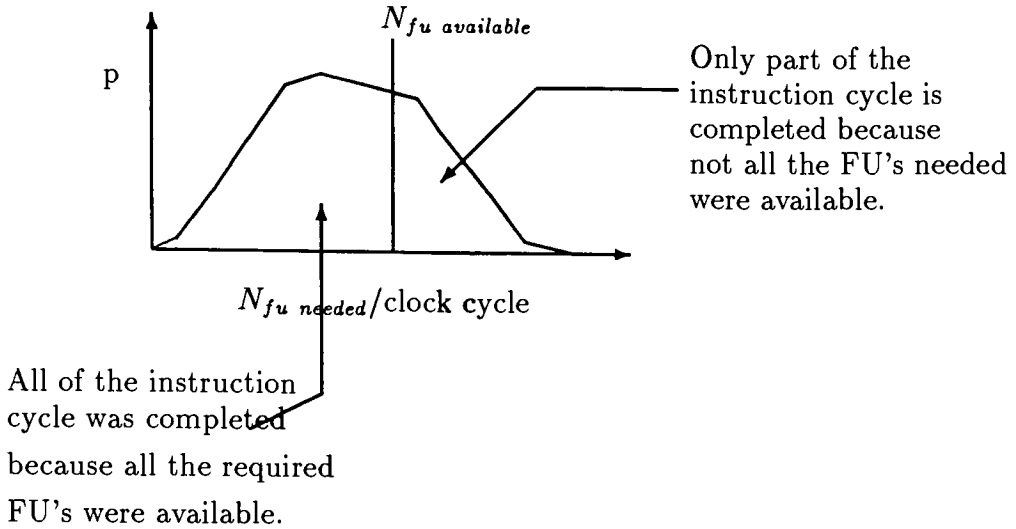


Figure 2.5. Effect of Limited Functional Unit Resources on Run Length

2.3.3 Empirical Analysis of Speedup with Limited Resources

In this part of the analysis, realizable configurations with limited resources are simulated to determine the actual speedup. The same configurations will also be applied to the analytical model proposed in the previous section, and the results are compared.

To completely test the analytical model, the "cc" compiled Dhrystone benchmark is empirically analyzed over a full spectrum of instruction window sizes, and functional unit configurations. This test program was selected for full analysis because its synthetic nature allows it to provide the "average" view of a program with the minimum number of traced instructions.

Figure 2.6 details the effect of window size on speedup. In this set of simulations the processor is allowed to have infinite functional unit resources, so that only the effect of limited window resources are shown. Also shown is the speedup as predicted by the analytical model.

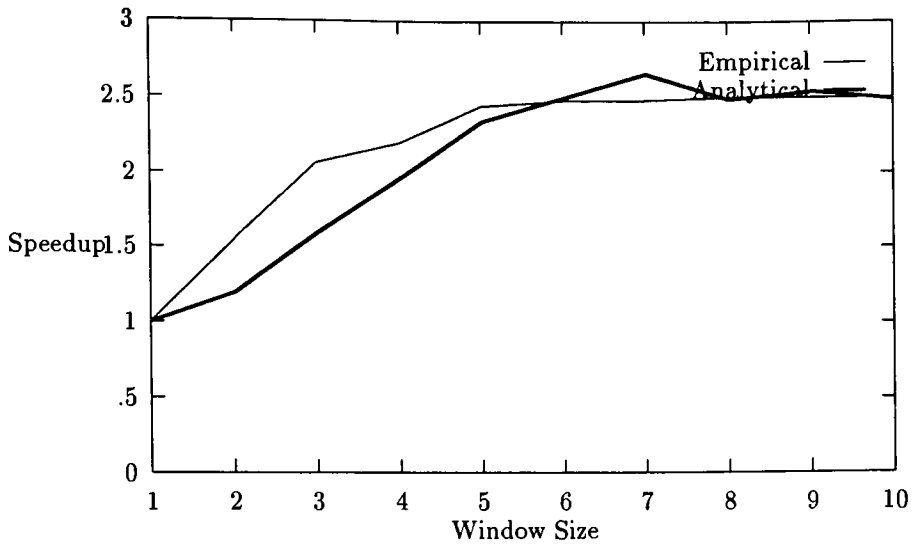


Figure 2.6. Speedup vs. Window Size for Small Run Lengths

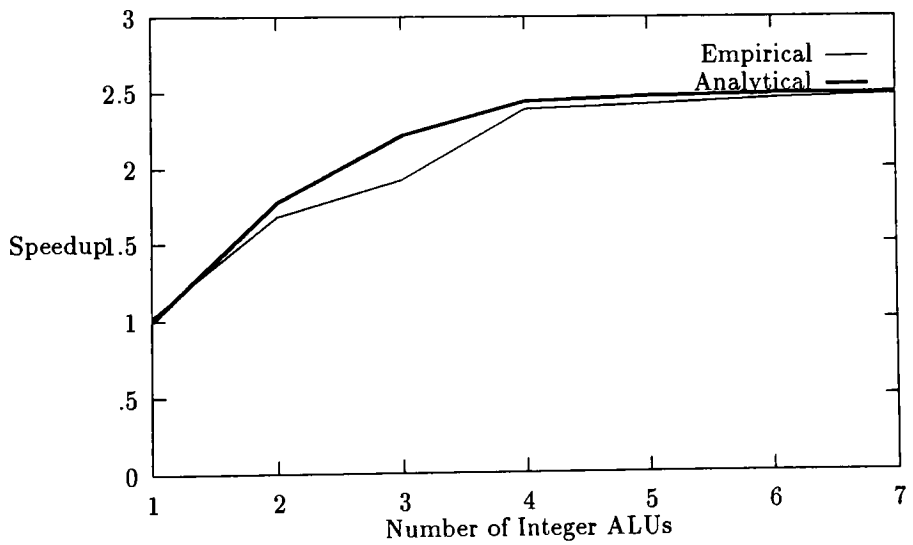


Figure 2.7. Speedup vs. Number of Integer ALUs

Figure 2.7 details the effect of integer ALUs resources on speedup. In these simulations, the simulated processors have infinite window resources, so only the effects of limited functional unit resources are shown. Also shown is the speedup as predicted by the analytical model.

Appendix B details the algorithms used to simulate the instruction issue logic, and the algorithm used to measure needed simulator characteristics.

For this program, window sizes larger than 5 instructions provide very little speedup. This is an interesting result since the expected run length of the optimal analysis is slightly over 7 instructions. This implies that the parallelization is mostly localized such that a 5 instruction wide instruction window can exploit most of the implicit parallelism. With an expected run length of about 7 instructions there is probably only one operation per run resulting in only localized non-dependencies.

The predicted speedup results indicates that the analytic model works well only for the larger window sizes. The reasons for this are further investigated in Section 2.3.4.

For integer ALU resources, any more than four provide very little benefit to this program. As derived in the analytical model, the fraction of functional unit resources required compared to the optimal number of arithmetic logic units needed determines the amount of speedup that will be realized. For the "cc" based Dhrystone program, the expected number of integer ALUs per cycles is about 2.4. With this many integer ALUs, the processor will fulfill one half of all the processor integer ALU requirements. If only one half of the instruction issues have enough resources, it is expected that the resulting speedup will be approximately one half of the optimal speedup. Figure 2.7 has a speedup of approximately 1.75 which is the half way point between speedup of 1.0 and 2.5, the nonparallelized and optimal speedups respectively. Sources of error arise when instructions are pushed from a cycle with already one instruction to another cycle with one instruction that can

fulfill the resource requirements. This situation results in a smoothing of the peak resources cycles to other nearby cycles where resources might have gone unused in the optimal analysis. The resulting compression for the run is unaffected by this phenomenon. The analytical model performed very well over all ranges of functional units resources.

A number of other measurements are available during resource limited simulation. These will help determine how well the configuration's hardware is utilized. These measurements include:

- Program Speedup.
- Instruction Window Utilization. This is the average fraction of the window that is filled for any cycle.
- Functional Unit Utilization. This is the probability that a functional unit will be issued an instruction for any given clock cycle.

To reduce the test space from which to choose, three interesting configurations were chosen and simulated. The configurations consist of:

- A configuration with two integer ALUs, two floating point ALUs, and one of each other functional unit. A window size of four instructions will be used. This configuration will be able to parallelize instructions that require the same type of ALU, although not to the extent needed to fully exploit the implicit parallelism available. (labeled Minimum configuration)
- A configuration that will attain at least 60% of the functional unit and instruction look ahead requirements of all programs traced. This configuration has four integer ALUs, four FP ALUs, and one of all other functional units. It has a window size of 8 instructions. (labeled Medium configuration)

- A configuration that will satisfy at least 90% of all the functional unit and instruction look ahead requirements of all programs traced. This configuration has seven integer ALUs, six FP ALUs, and one of all other functional units. It has a window size of 16 instructions. (labeled Maximum configuration)

The results of the resource limited speedup is shown in Tables 2.14, 2.16, and 2.18. Tables 2.15, 2.17, and 2.19 show the functional unit requirements of each simulation.

Table 2.14. Speedup Results for the Minimum Configuration

Program	Actual Speedup	Percent of Optimal	Predicted Speedup	Percent Error
Dhrystone (cc)	1.619880	64.8	1.393978	13.9
Dhrystone (gcc)	1.528639	67.9	1.682054	10.0
Dhrystone (MESA)	1.491449	79.2	1.298630	12.9
Dhrystone (ADA)	1.613430	84.5	1.457901	09.6
LINPACK	1.864634	66.1	2.109876	13.2
C-Prolog	1.604149	68.8	1.490092	07.1
nroff	1.468597	83.2	1.409815	04.0
FBM Image Halftone	1.453366	80.5	1.407463	03.2
XfRoot	1.614204	71.5	1.328338	17.7
Invert	1.958588	52.1	2.243482	14.5
gcc compile	1.489752	76.1	1.380406	07.3
grep	1.503554	84.9	1.361571	09.4

Table 2.14 shows the speedup results for the Minimum configuration. All the programs attained at least 50% of the optimal speedup, with the less resource hungry programs, such as nroff and grep, attaining as much as 84.5% of optimal speedup. If the Minimum configuration could be built with twice as much hardware as the present, non-parallelizing configurations, the processor would give as much as linear speedup for the hardware invested. This is comparable to explicitly parallel architectures. Unfortunately, these rewards diminish very quickly.

Table 2.15. Resource Utilization for the Minimum Configuration

Program	Window Utilization	Functional Unit Utilization
Dhrystone (cc)	95.1	Int ALU: 99.9 58.1 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
Dhrystone (gcc)	95.5	Int ALU: 99.9 46.1 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
Dhrystone (MESA)	95.1	Int ALU: 99.8 44.7 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
Dhrystone (ADA)	90.9	Int ALU: 99.9 42.0 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
LINPACK	96.7	Int ALU: 87.3 27.3 FP ALU: 35.5 25.4 FP Mult: 06.8 FP Div: 00.6 FP Srqt: 00.0
C-Prolog	96.8	Int ALU: 99.9 53.6 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
nroff	94.3	Int ALU: 99.8 41.1 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
FBM Image Halftone	94.9	Int ALU: 99.7 41.0 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
XfRoot	96.7	Int ALU: 94.9 38.8 FP ALU: 14.7 04.9 FP Mult: 02.8 FP Div: 00.6 FP Srqt: 00.0
Invert	98.3	Int ALU: 71.4 33.8 FP ALU: 47.5 28.4 FP Mult: 12.1 FP Div: 00.6 FP Srqt: 00.0
gcc compile	95.6	Int ALU: 99.9 43.5 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
grep	92.1	Int ALU: 99.9 34.5 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0

Table 2.15 details the resource utilization for each test program executing on the Minimum configuration simulator. Window utilization is the average number of window slots filled each cycle as a percentage. The functional unit utilization is the probability of the functional unit being dispatched an instruction for any clock cycle. Functional unit utilizations values are shown for each type of functional unit, and for each functional unit if there are more than one.

Table 2.16. Analysis Results for the Medium Configuration

Program	Actual Speedup	Percent of Optimal	Predicted Speedup	Percent Error
Dhrystone (cc)	2.404274	96.2	2.419806	00.6
Dhrystone (gcc)	2.016466	89.7	2.129550	05.6
Dhrystone (MESA)	1.806023	95.9	1.808919	00.2
Dhrystone (ADA)	1.890766	99.0	1.826308	03.4
LINPACK	2.414125	85.6	2.189622	09.3
C-Prolog	2.091788	89.7	2.303375	10.1
nroff	1.711489	96.9	1.757043	02.7
FBM Image Halftone	1.735283	96.2	1.739083	00.2
XfRoot	2.092353	92.8	2.080505	00.6
Invert	2.953491	78.5	3.131980	06.0
gcc compile	1.881977	96.2	1.947181	03.5
grep	1.749797	98.9	1.706614	02.5

Table 2.16 shows that the Medium configuration provides almost optimal speedup for a number of the less resource intensive programs. It also provides well over 70% of the optimal speedup for all programs, including the resource intensive math software.

Table 2.18 shows speedup for the Maximum configuration. High resource utilization is realized by all the test programs. Yet, speedup opportunities are still available for the mathematically intensive programs, such as LINPACK, and Invert.

Table 2.17. Resource Utilization for the Medium Configuration

Program	Window Utilization	Functional Unit Utilization
Dhrystone (cc)	74.7	Int ALU: 99.8 63.1 43.5 28.1 FP ALU: 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
Dhrystone (gcc)	79.9	Int ALU: 99.9 40.6 32.1 20.1 FP ALU: 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
Dhrystone (MESA)	74.4	Int ALU: 99.7 44.0 22.2 09.1 FP ALU: 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
Dhrystone (ADA)	69.7	Int ALU: 99.9 45.1 17.4 03.8 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
LINPACK	79.7	Int ALU: 92.7 38.4 09.9 03.1 FP ALU: 29.1 25.0 14.5 14.4 FP Mult: 09.4 FP Div: 00.7 FP Srqt: 00.0
C-Prolog	85.7	Int ALU: 99.9 56.5 27.3 16.5 FP ALU: 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
nroff	72.1	Int ALU: 99.8 39.7 17.3 07.3 FP ALU: 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
FBM Image Halftone	75.7	Int ALU: 99.6 39.6 20.9 07.7 FP ALU: 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
XfRoot	83.2	Int ALU: 98.4 43.1 22.2 09.6 FP ALU: 13.6 05.8 04.0 01.9 FP Mult: 03.6 FP Div: 00.8 FP Srqt: 00.0
Invert	93.4	Int ALU: 78.2 44.2 24.9 11.4 FP ALU: 51.9 30.5 18.4 13.4 FP Mult: 18.2 FP Div: 00.9 FP Srqt: 00.0
gcc compile	76.7	Int ALU: 99.9 44.8 25.7 10.7 FP ALU: 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
grep	64.9	Int ALU: 99.9 42.8 16.4 03.1 FP ALU: 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0

Table 2.18. Analysis Results for the Maximum Configuration

Program	Actual Speedup	Percent of Optimal	Predicted Speedup	Percent Error
Dhrystone (cc)	2.494091	99.8	2.587694	03.8
Dhrystone (gcc)	2.248601	99.9	2.253961	02.4
Dhrystone (MESA)	1.878078	99.8	1.917554	02.1
Dhrystone (ADA)	1.904663	99.7	1.864063	02.1
LINPACK	2.530923	89.7	2.389319	05.6
C-Prolog	2.272154	97.5	2.459660	08.3
nroff	1.764227	99.9	1.780717	00.9
FBM Image Halftone	1.802894	99.9	1.695454	06.0
XfRoot	2.202503	97.7	2.275319	03.3
Invert	3.212528	85.4	3.584439	11.6
gcc compile	1.949364	99.6	1.959199	00.5
grep	1.763719	99.7	1.769131	00.3

In all configurations, only one floating point multiplier, divider, and square root hardware is needed. This is consistent with the results from the previous analysis indicating that very rarely could more than one of these operations be dispatched at a time.

2.3.4 Accuracy of the Analytical Model

As shown in the previous section, the analytic model does very well for predicting the results of the Maximum configuration, acceptably for the Medium configuration, and marginally for the Minimum configuration. No predicted speedup error exceeded 20%. Figures 2.6 and 2.7 indicate the the source of predicted error is mostly from the interpolation of speedup due to limited instruction window resources. This error is larger for smaller window sizes. The smaller window sizes rely more heavily on interpolation of run length expansion due to run lengths that do not fit entirely into the instruction window. This indicates that the models method of

Table 2.19. Resource Utilization for the Maximum Configuration

Program	Window Utilization	Functional Unit Utilization
Dhrystone (cc)	50.1	Int ALU: 99.8 63.0 42.7 28.5 05.3 02.8 01.1 FP ALU: 00.0 00.0 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
Dhrystone (gcc)	53.8	Int ALU: 99.9 43.8 35.4 22.6 12.1 00.6 00.4 FP ALU: 00.0 00.0 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
Dhrystone (MESA)	42.2	Int ALU: 99.7 42.7 21.5 09.3 04.9 02.9 00.9 FP ALU: 00.0 00.0 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
Dhrystone (ADA)	41.1	Int ALU: 99.9 43.4 17.1 04.2 01.5 00.6 00.6 FP ALU: 00.0 00.0 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
LINPACK	62.7	Int ALU: 92.1 37.8 10.4 03.2 01.1 00.2 00.0 FP ALU: 30.9 26.9 17.0 17.0 00.5 00.4 FP Mult: 10.8 FP Div: 00.7 FP Srqt: 00.0
C-Prolog	67.5	Int ALU: 99.9 55.0 25.2 16.6 11.2 06.6 03.0 FP ALU: 00.0 00.0 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
nroff	39.7	Int ALU: 99.8 39.2 17.3 07.8 03.1 01.5 00.5 FP ALU: 00.0 00.0 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
FBM Image Halftone	45.5	Int ALU: 99.6 38.8 20.7 07.4 04.6 02.6 00.7 FP ALU: 00.0 00.0 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
XfRoot	58.7	Int ALU: 98.3 43.1 22.0 09.2 05.3 03.1 01.3 FP ALU: 12.2 06.7 03.8 01.7 01.3 01.1 FP Mult: 03.8 FP Div: 00.8 FP Srqt: 00.0
Invert	83.9	Int ALU: 79.3 47.4 26.2 11.0 04.9 02.4 00.9 FP ALU: 46.7 28.8 15.7 14.1 09.8 09.6 FP Mult: 19.9 FP Div: 00.9 FP Srqt: 00.0
gcc compile	45.3	Int ALU: 99.9 44.3 25.3 11.1 04.2 01.9 00.9 FP ALU: 00.0 00.0 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0
grep	33.5	Int ALU: 99.9 41.8 16.3 03.1 02.3 00.8 00.1 FP ALU: 00.0 00.0 00.0 00.0 00.0 00.0 FP Mult: 00.0 FP Div: 00.0 FP Srqt: 00.0

interpolation is not entirely accurate. The model currently uses the compression characteristics of runs the same size as the instruction window to predict the number of cycles that will be required to completely execute the entire uncompressed run. Unfortunately, the number of samples used to determine the compression characteristic are significantly less for the shorter runs.[3] This results in even greater errors when predicting speedup for very limited resource configurations. Only one node of the graph shows any noticable error.

The effects on speedup due to limited functional units is very accurate, as shown in Table 2.7.

2.3.5 Memory Bandwidth Requirements

When a parallel processor is executing, it will need to fill its instruction window so that it may select instructions to execute. This process requires a significant amount of bandwidth since the processor doesn't fetch single instructions but rather runs.

Figures 2.8, 2.9, and 2.10 shows the memory bandwidth requirements of the SunOS "cc" compiled Dhrystone benchmark for the three studied configurations.

For all the configurations, bandwidth bursts as high as the window size are required. All configurations also have a large fraction of cycles that do not require any instruction fetches. For the maximum configuration, 60% of the cycles require no instruction fetches, and most others require 6, 8, or 15 instructions to be fetched in one cycle.

A memory system that could fetch 15 instruction in one cycle would be very complicated and expensive. It would probably have to be an interleaved memory system, and be able to quickly resolve situations where memory values resided in the same memory unit. Much research will be required to build efficient memory

[3]See Figure 2.2 for an example of this.

Figure 2.8. Bandwidth Requirements for the Minimum Configuration

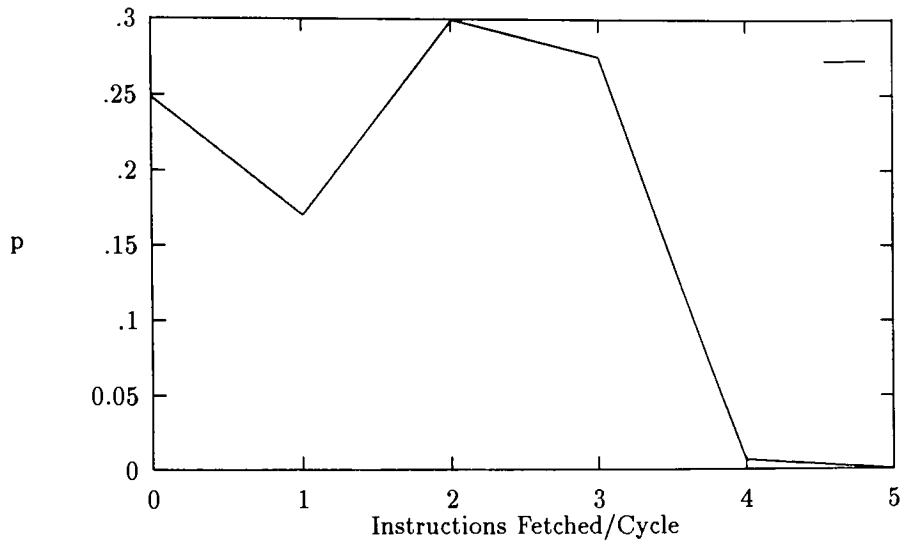


Figure 2.9. Bandwidth Requirements for the Medium Configuration

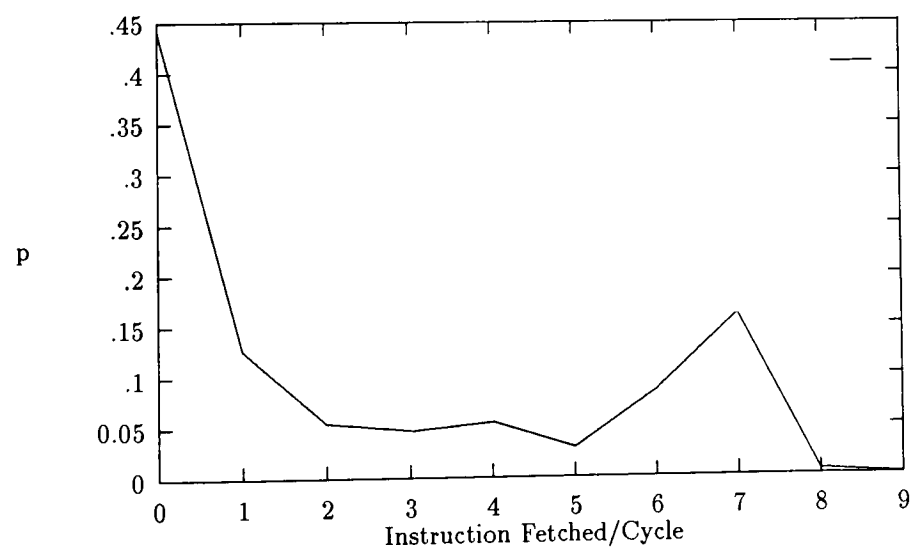
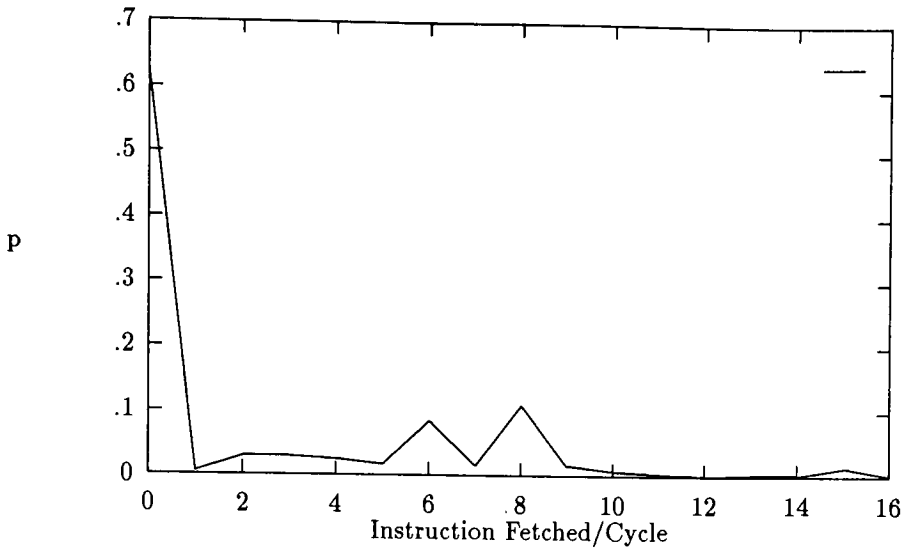


Figure 2.10. Bandwidth Requirements for the Maximum Configuration



systems for parallelizing processors. A clever implementation could probably use the cycles in which no instruction fetches are required to fetch ahead of the program counter. If conditional branches are encountered, fetching could continue down both paths until the conditional branch was resolved.

2.4 Conclusions

In this thesis, it was shown that exploiting implicit parallelism on the SPARC architecture can provide speedup from 50% to nearly 400%. The question of compiler optimization and code generation technology was discussed with respect to its effect on speedup. It was shown that optimization can reduce the amount of implicit parallelism available to a parallelizing processor. Given the optimal trace characteristics an analytical model was developed to predict the effects of limited speedup on the optimal results. The model performed best for configurations that were not extremely resource limited. It was suggested that this inaccuracy was primarily due to errors in the interpolation on the effects of limited window size on run length compression. Next, three configurations were chosen that characterized

theoretical processors. The first could be realized with a minimum amount of hardware, while the others supplied more resources to the simulated processor. The third configuration supplied programs with nearly all the resources required, thus attaining nearly optimal speedup for all simulations. Lastly, simulation characteristics provided a view of the memory bandwidth requirements for a parallelizing processor. It was shown that the reference patterns of these processors are very bursty, often needing to fetch an entire window of instructions in one clock cycle.

This thesis highlights a number of directions in which further research could be directed.

- Although the speedups attained in the simulations of this thesis are worthwhile, further speedup could be attained if independencies could be exploited between basic blocks.

- Currently, very few parallelizing processors have been developed. This is primarily due to the complexity of the hardware needed to do the instruction issue and the large number of transistors needed to implement multiple functional units. Better hardware designs for both instruction issue logic and functional units could speed these processors to market.

- It was shown how optimization and implicit parallelism can conflict. Much research is needed into optimization of code for parallelizing architectures.

- It was shown that the memory reference patterns of parallelizing processors are very bursty. Much research will be needed to develop memory systems that will satisfy the hungry demands of parallelizing processors.

- The analytical model can provide an excellent design tool for the computer architect. The one developed in this thesis suffered from inaccuracies in predicting changes in speedup due to limited window resources. A better model for this process would make the analytical model more useful.

APPENDIX A

SPARCTRACE USERS MANUAL

SPARCTrace is a program that has been developed for capturing SPARC processor instruction traces. It is a general purpose trace tool that creates execution traces with sufficient information such that a simulator could later re-execute the trace. The tool is implemented on SunOS 4.x, thus it will only execute on machines that share binary compatibility with the Sun SPARC workstation series.

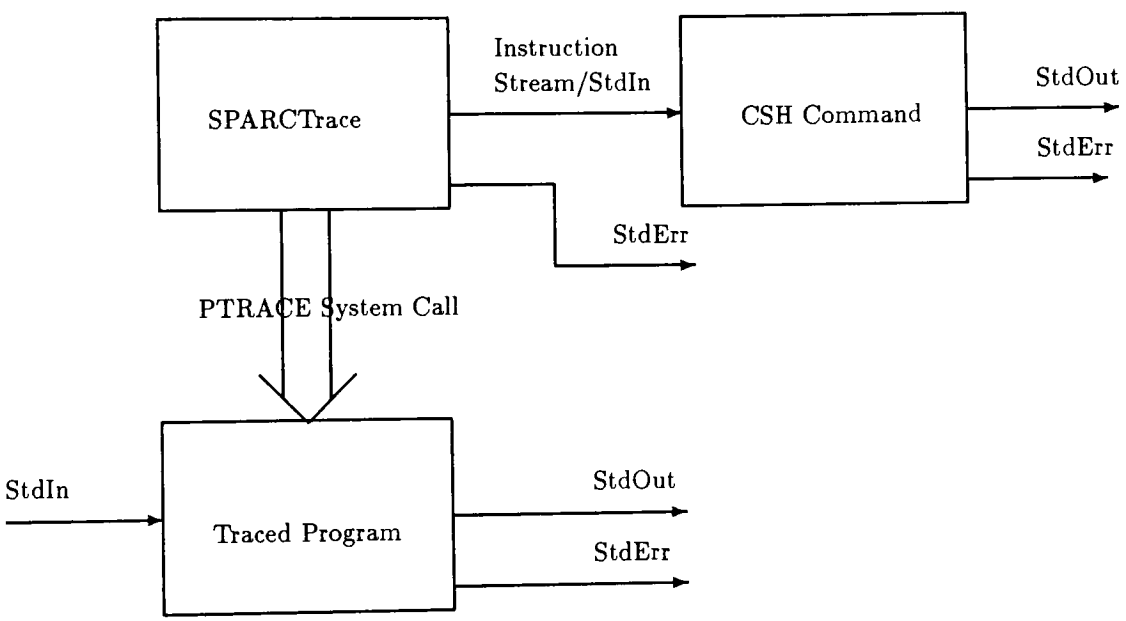


Figure A.1. SPARCTrace Operation

Figure A.1 details operation of the SPARCTrace program. The traced program runs without knowledge that it is being traced. Its StdIn, StdOut, and StdErr file streams remain in tact. SPARCTrace uses the Unix PTRACE system call to

impose its control over the program being traced. Any SPARCTrace informational output is inserted into the StdErr stream of the program being traced. Instruction stream information is piped to another C-Shell's StdIn stream. A SPARCTrace command line option allow the user to specify the C-Shell command line that will accept the instruction trace stream on its StdIn stream.

SPARCTrace accepts the following command line format:

```
SPARCTrace [options...] "csh_command" traced_command [args...]
```

The `[options...]` are all optional and can be one of the following:

```
-c n        Dump checkpoint information every n instructions.
-d          Dump debug information to StdErr.
```

The `"csh_command"` argument is required and specifies the first command to execute in a new C-Shell that will accept the instruction trace stream from StdIn.

The `traced_command` argument is the name of a programs to trace, and `[args...]` is any number of optional arguments for the `traced_command`.

For example;

```
SPARCTrace -d 1000 "compress > /dev/rst0" grep foo *
```

This command will trace the command `grep foo *`. All trace output will be piped to compress and then redirected to a tape device.

```
SPARCTrace "SPARCAnalyze > results.txt" cc -o main main.c
```

This command will trace the command "cc -o main main.c". All trace output is piped directly into a program called SPARCAnalyze, whose output is put into the file "results.txt".

APPENDIX B

ALGORITHMS USED

B.1 Algorithms Used for Optimal Speedup Analysis

The following algorithm is used to determine optimal speedup in SPARC instruction traces.

```
initialize counters
open instruction stream
WHILE more instructions still in the stream DO
  fetch instructions to the next delimiter
  record pre-compressed run stats
  compress the fetched run
  record post-compressed run stats
  execute the compressed run
ENDWHILE
dump final statistics
```

The following algorithm is used to compress a run.

```
FOR currentInstruction = second TO last instruction DO
  WHILE prevInstruction exists AND
        NotDepend(prevInstructions, currentInstruction) DO
    move currentInstruction to previous clock cycle
  ENDWHILE
ENDFOR
```

The dependency between two instructions are determined with the following algorithm.

```
domainA = referents of instruction A
rangeA = side effects of instruction A
domainB = referents of instruction B
rangeB = side effects of instruction B
dependent if: ((rangeA AND domainB) != NULL SET) OR
((rangeB AND domainA) != NULL SET) OR
((rangeA AND rangeB)  != NULL SET)
```

All slot management is implemented via link list structures. All the slots are link into a list. And each slot entry consists of a list of instructions currently residing in that slot.

B.2 Algorithms Used for Resource Limited Speedup Analysis

The following algorithm is used to determine resource limited speedup in SPARC instruction traces.

```
initialize counters
read resource configuration parameters
open instruction stream
WHILE more instructions still in the stream DO
  release all functional units that have completed
  fetch instructions UNTIL window full OR delimiter encountered
  record pre-compressed run stats
  compress the current run
  record pre-compressed run stats
```

```
  execute the next clock cycle
ENDWHILE
dump final statistics
```

Run compression is the same as in the optimal analysis case. Execution however is changed because resource limitations not exist.

```
FOR all instruction in the currentSlot DO
  IF currentInstruction resources available THEN
  BEGIN
    execute the current instruction
    indicate that the functional unit is in use
  END
ENDFOR
```

The functional units do not always need one cycle to complete execution, so they are tagged with a counter that will indicate which clock cycle the functional unit will become free. The main loop of the execution algorithm frees the functional units when there operations have completed.

REFERENCES

[1] Ramon D. Acosta, Jacob Kjelstrup, and H. C. Torng, *An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors*, IEEE Transactions on Computers, September 1986.

[2] R. M. Tomasulo, F. J. Sparacio, and D. W. Anderson, *The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling*, IBM Journal, January 1967.

[3] R. M. Keller, *Look-ahead Processors*, Computer Survey, December 1975.

[4] J. E. Thorton, Design of a Computer: The Control Data 6600, Published by Scott, Foresman and Co., 1970.

[5] Richard M. Russel, *The CRAY-1 Computer System*, Communications of the ACM, January 1978.

[6] G. S. Sohi and A. R. Pleszkun, *The Performance Potential of Multiple Functional Unit Processors*, Conference Proceedings from the 15th Annual International Symposium on Computer Architecture, May 1988.

[7] Reinhold P. Weicker, *The "Dhrystone" Benchmark Program*, Communications of the ACM, October 1984.

[8] Sun Microsystems, The SPARC Architecture Manual, Published by Sun Microsystems, Inc., June 1987.

[9] Robert B. Garner, et al, *The Scalable Processor Architecture (SPARC)*, 1988 CompCon Proceedings, May 1988.

[10] D. Patterson, *Reduced Instruction Set Computers*, Communications of the ACM, January 1985.

[11] D. Patterson and C. Sequin, *RISC 1: A Reduced Instruction Set VLSI Computer*, Conference Proceedings from the 8th Annual International Symposium on Computer Architecture, May 1981.

[12] R. Blau Unger, P. Foley, A. Samples, and D. Patterson, *Architecture of SOAR: Smalltalk on a RISC*, Proceedings of the 11th Annual Symposium on Computer Architecture, June 1984.

[13] M. Richardson, *Instruction Issue Logic in the IBM 6000 Workstation*, IBM Journal, January 1990.

[14] G. S. Sohi, *Instruction Issue Logic for High Performance, Interruptable Pipelined Processors*, Conference Proceedings from the 14th Annual Symposium on Computer Architecture, June 1987.

[15] S. Vajapeyam, G.S. Sohi, W. Hsu, *Exploitation of Operation-Level Parallelism in a Processor of the CRAY X-MP*, Proceedings of the ICCD, 1990.