UNIVERSITY OF CALIFORNIA

Irvine

Performance Analysis of a Superscalar Architecture

A thesis submitted in partial satisfaction of the

requirements for the degree Master of Science

in Engineering

by

Steven Daniel Wallace

Committee in charge:

Professor Nader Bagherzadeh, Chair

Professor Daniel Gajski

Professor Fadi Kurdahi

1993

The thesis of Steven Daniel Wallace is approved:

Committee Chair

University of California, Irvine

1993

Dedication

To my parents,
for their never-ending love and support.

Contents

List of Figures

List of Tables

Acknowledgements

I give special thanks to Professor Nader Bagherzadeh, my advisor, for his guidance and support.

I thank my thesis committee members, Professor D. Gajski and Professor F. Kurdahi for their reading and evaluation of my thesis work. I also thank Marcelo Moraes De Azevedo for his comments and Manu Gulati for his proof-reading.

I thank John Lenell for teaching me the basics of a superscalar microprocessor.

I thank all members of the SDSP research group and fellow graduate students in the lab for their encouragement. Especially, I thank Sheng Huang for use of some of his figures and for helping me get the drawing program print my figures.

I also appreciate the Chancellor's Fellowship from UC Irvine.

Abstract of the Thesis

Performance Analysis of a Superscalar Architecture

by

Steven Daniel Wallace

Master of Science in Engineering

University of California, Irvine, 1993

Professor Nader Bagherzadeh, Chair

The Superscalar Digital Signal Processor (SDSP) architecture is presented. Decode size, fetch efficiency, branch prediction, dynamic scheduling, resource allocation, issue rate, result write rate, and cache are discussed. Over 50 *billion* simulated cycles on seventeen different programs were run on over a dozen different configurations. Notably, results show that the average instruction fetch places an upper bound on speedup and is the most critical factor in determining overall performance. In addition, a new approach in handling branch prediction is shown to significantly decrease the bad branch penalty. Furthermore, increasing the issue limit beyond the decode size is shown to improve performance, but for maximum improvement, the rate at which results are written should also be increased. Overall, the speedup for the SDSP is over two and a half times faster than an ideal scalar version.

Chapter 1

Introduction

For many years, the route to good performance was by demanding more functionality out of each individual instruction. This was accomplished using a CISC (Complex Instruction Set Computer) architecture. Then in the 1980's, a different approach, RISC (Reduced Instruction Set Computer), was effectively used to outperform a comparable CISC processor. By using a simple instruction set, a RISC designer can use aggressive hardware techniques, such as pipelining [9], to reduce the cycle time and the number of cycles per instruction, thereby increasing overall performance. Today, most RISC processors typically execute one cycle per instruction. A plateau in performance improvement with the basic RISC processor has been reached; therefore, alternative or additional techniques need to be explored to improve performance.

In recent years, there has been a growing interest in designing microprocessors based on the notion of Instruction-Level Parallelism (ILP). There are different approaches for exploiting ILP. One approach uses run-time scheduling to evaluate the data dependencies and execute instructions concurrently. A microprocessor based on this technique is called a superscalar microprocessor [8]. Another approach, commonly known as a Very Long Instruction Word (VLIW) architecture [5], is entirely based on compile-time analysis to extract parallelism.

Superscalar architectures and VLIW architectures both improve processor performance by reducing the CPI (Cycles Per Instruction) factor. These architectures exploit ILP by issuing more than one instruction per cycle. VLIW processors require a sophisticated compiler to extract instruction-level parallelism prior to execution which results in code expansion [12, 4]. Superscalar architectures utilize dynamic scheduling to extract parallelism at run-time, in addition to static scheduling [3, 8]. This advantage, however, is gained at the expense of significant hardware complexity.

1.1 Performance Limitations

A processor capable of executing more than one instruction in a single cycle must fetch multiple instructions from memory each cycle. A processor that fetches n instructions per cycle will ideally have a speedup of n over a scalar processor. Hazards, however, limit the size of n, depending on the architecture. These three basic hazards are [6]:

- *Structural hazards*: A conflict between multiple instructions which require the same resource at the same time.
- *Data hazards*: An instruction depends on the result of a previous instruction.
- *Control hazards*: Instructions following a branch cannot be executed until the branch is resolved.

Hazards can make the processor stall and cause the processor not to reach ideal performance.

1.2 Superscalar Processors

Superscalar processors fetch, issue, and execute multiple instructions in a single cycle. Functional units which execute the instruction may be of different types, duplicates, or a combination.

There are three main policies for issue and completion of instructions:

In-order issue and completion: Instructions are issued in their original program order and results are written in the same order. All scheduling halts if an instruction is waiting for results of a previous instruction, so independent instructions which may have their operands ready cannot be issued.

In-order issue with out-of-order completion: Instructions are issued in their original program order, but results may be received out-of-order. This is useful for a long-latency load or floating-point operation [14]. This allows instructions that do not have a structural or data hazard to execution without waiting.

Out-of-order issue and completion: Out-of-order issue is used to look beyond instruction dependencies found with in-order issue. An instruction window is used to look at a pool of instructions from which independent instructions are selected for execution. The order of issue and completion may be different from the original program order.

Out-of-order issue and completion requires that the state of the register file be maintained, which can be accomplished using several different methods:

- *Register scoreboard:* Each register indicates if there is a pending update [16, 18].
- *Future file:* Only the most recent value is allowed to be written into a future file. It must keep track of the most recent update and check its status before writing.
- *Reorder buffer:* Renames each destination register instance to a unique identifier. An associative lookup maps this identifier to the entry when results are written. The values are then written back to the register file and the identifier is discareded [15].

Performance is limited by instruction dependencies and sufficient instruction bandwidth. Instruction dependencies are reduced by a dynamic instruction scheduler in an instruction window. Two main methods exist to implement an instruction window:

Reservation stations: Reservation stations partition the instruction window by functional unit. Only instructions at each station are considered for scheduling on its assigned functional unit, which simplifies control logic. The algorithm was first developed by Tomasulo in 1967 [17].

Central instruction window: A central instruction window keeps all unissued instructions in one unit, regardless of the type of instruction [8]. It determines which ready-to-run instructions are scheduled onto what functional unit, thereby complicating control logic.

1.3 Scope of Thesis

This thesis analyzes the performance issues of a superscalar processor, the SDSP (Superscalar Digital Signal Processor) developed at the University of California, Irvine. The SDSP uses a centralized window and dynamic register renaming to allow out-of-order issue and completion of instructions. The following chapter explains the SDSP architecture in detail.

Chapter 3 discusses the tools and methods used for simulation: the compiler, assembler, linker, simulator, and test programs. The GNU CC compiler was ported along with the GNU GAS assembler and a GNU linker. A simulator was developed using the C language to execute SDSP object code on a SUN workstation. The simulator is configurable for many different parameters so it can evaluate the performance *accurately* via execution-driven methods. After collecting results from different runs of several programs, the performance is evaluated to determine which areas impede and improve performance.

Finally, Chapter 4 presents the results from simulation. Decode size, instruction fetch efficiency, branch prediction, scheduling, cache, and resource allocation are all considered for their impact on performance.

Chapter 2

The Superscalar Digital Signal Processor Architecture

The Superscalar Digital Signal Processor (SDSP) is a 32-bit superscalar processor with a RISC-style instruction set. It is pipelined and has a fetch bandwidth of four instructions per cycle. The SDSP research project began in 1992 at U.C. Irvine [12]. The processor is especially suited for digital signal processing applications, yet is powerful on any general purpose application. It follows VLSI design strategies of VIPER and TinyRISC architectures [1, 5]. Design decisions were made with full consideration for VLSI implementation. In addition, a VHDL model has already been developed [13]. The VLSI design and VHDL modeling contribute to the evaluation of design tradeoffs and performance issues. This chapter presents the SDSP architecture in detail, by first defining its instruction set and then explaining its architectural organization.

2.1 Instruction Set

The SDSP uses a fixed 32-bit format for its instruction set. It has 32 general purpose registers. The instructions set is divided into three classes, each of which has a different instruction format. The two most significant bits of an instruction identify its class. Figure 2.1 shows the format of the three instruction classes.

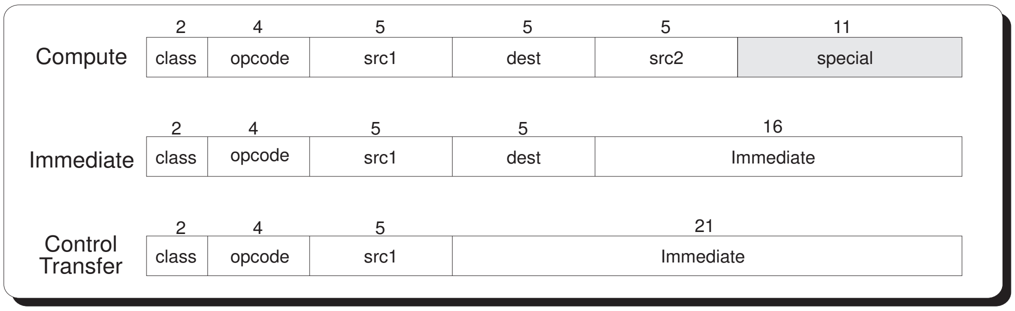

Figure 2.1: SDSP Instruction Format

2.1.1 Compute Class

The compute class format consists of integer arithmetic, logic, shift, load, and store operations. All instructions in this group use registers as operands. The instruction format is subdivided into class, opcode, source register 1 (*src1*), source register 2 (*src2*), and destination register (*dest*). It also has a special field which must be set to zero (reserved for future expansion). Table 2.1 lists all the compute instructions. Currently, the SDSP implementation does not have an integer divide unit, so divide instructions are emulated via a trap to a special location. Otherwise, hardware support must be provided to support this instruction.

Compute Instructions class = 00, special = 00000000000			
Opcode	Operation	Arguments	Description
0000	add	src1, src2, dest	*dest = src1 + src2*
0001	sub	src1, src2, dest	*dest = src1 - src2*
0010	lsl	src1, src2, dest	*dest = src1 << src2*
0011	lsr	src1, src2, dest	*dest = (unsigned) src1 >> src2*
0100	sge	src1, src2, dest	*dest =* $(src1 \geq src2)$ *? -1 : 0*
0101	sgeu	src1, src2, dest	*dest = (unsigned)* $(src1 \geq src2)$ *? -1 : 0*
0110	seq	src1, src2, dest	*dest = (src1 == src2) ? -1 : 0*
0111	asr	src1, src2, dest	*dest = src1 >> src2*
1000	and	src1, src2, dest	*dest = src1 AND src2*
1001	or	src1, src2, dest	*dest = src1 OR src2*
1010	xor	src1, src2, dest	*dest = src1 XOR src2*
1011	lu	src1, src2, dest	*dest = src1[0...16] | src2 << 16*
1100	mul	src1, src2, dest	*dest =* $src1 \times src2$
1101	div	src1, src2, dest	*dest =* $src1 \div src2$
1110	ldw	src1, src2	*src1 = M[src2]*
1111	stw	src1, src2	*M[src2] = src1*

Table 2.1: Compute Instructions

2.1.2 Compute-Immediate Class

The Compute-Immediate class performs the same operations as the Compute class. These instructions use an immediate value instead of a register operand for the *src2* field. The instruction format is subdivided into class, opcode, *src1* register, *src2* 16-bit (I16) or 21-bit (I21) immediate, and *dest* register. The immediate operand is sign-extended (SE) or zero-extended (ZE). The *ldw* instruction uses the *src1* field instead of the *dest* field so that the 21 bit immediate address field is contiguous. Note, the immediate address is a *word* address rather than a byte address so no bit fields are wasted. Table 2.2 lists all compute-immediate instructions.

Compute Immediate Instructions class = 01			
Opcode	Operation	Arguments	Description
0000	add	src1, I16(ZE), dest	*dest = src1 + I16*
0001	sub	src1, I16(ZE), dest	*dest = src1 - I16*
0010	lsl	src1, I16, dest	*dest = src1 << I16*
0011	lsr	src1, I16, dest	*dest = (unsigned) src1 >> I16*
0100	sge	src1, I16(SE), dest	*dest = (src1* $\geq$ *I16) ? -1 : 0*
0101	sgeu	src1, I16(ZE), dest	*dest = (unsigned) (src1* $\geq$ *I16) ? -1 : 0*
0110	seq	src1, I16(SE), dest	*dest = (src1 == I16) ? -1 : 0*
0111	asr	src1, I16, dest	*dest = src1 >> I16*
1000	and	src1, I16(SE), dest	*dest = src1 AND I16*
1001	or	src1, I16(ZE), dest	*dest = src1 OR I16*
1010	xor	src1, I16(SE), dest	*dest = src1 XOR I16*
1011	lu	src1, I16, dest	*dest = src1[0...16]* $\mid$ *I16 << 16*
1100	mul	src1, I16(SE), dest	*dest = src1* $\times$ *I16*
1101	div	src1, I16(SE), dest	*dest = src1* $\div$ *I16*
1110	ldw	src1, I21	*src1 = M[PC[31...23]* $\mid$ *I21 << 2]*
1111	stw	src1, I21	*M[PC[31...23]* $\mid$ *I21 << 2] = src1*

Table 2.2: Compute-Immediate Instructions

2.1.3 Control Transfer Class

Instructions which involve transfer of control are grouped under the Control Transfer Class. This includes conditional and unconditional branches and trap instructions. Table 2.3 lists these instructions.

Conditional Branch Instructions

Branches may be taken or not taken based on the least significant bit of *src1*: branch true (*brt*) or branch false (*brf*). There is an immediate 21-bit field (I21) which determines the absolute address within the current PC segment. In addition, static branch prediction may be employed using branch true predict not taken (*brtnt*) or branch false predict not taken (*brfnt*). Otherwise, branches are predicted to be taken.

Control Transfer Instructions class = 10			
Opcode	Operation	Arguments	Description
0000	brf	src1, I21	*if (!src1[0]) PC[31...21] | I21*
0001	brt	src1, I21	*if (src1[0]) PC[31...21] | I21*
0010	brfnt	src1, I21	*if (!src1[0]) PC[31...21] | I21*
0011	brtnt	src1, I21	*if (src1[0]) PC[31...21] | I21*
0100	jump	src1	*PC = src1*
0101	call	src1	*R31 = PC; PC = src1*
0110	jumpi	I26	*PC = PC[31...28] | I26 << 2*
0111	calli	I26	*R31 = PC; PC = PC[31...28] | I26 << 2*
1000	trap	src1, src2,dest, I11	*trap to vector I11*
1001	trapos	I11	*trap to vector I11*
1110	rte		*return from exception*
1111	rti		*return from interrupt*

Table 2.3: Control Transfer Instructions

The implementation of this static branch prediction is optional, but all four branch opcodes must be handled.

Unconditional Branch Instructions

Control can be transferred unconditionally to a new Program Counter by using *jump*. The old PC can be saved to register 31 if the *call* instruction is used. Both types can accept either a source 1 register (*src1*) operand which contains a new 32-bit PC or a 26-bit immediate *word* address (I26).

Trap Instructions

When a trap instruction is encountered, the PC is saved into a special register, system mode bit is set, and the PC is changed to a pre-determined location dependent on the 11-bit immediate vector (I11). *Trapos* is intended for Operating System (OS) calls and parameters are passed using OS-defined registers, while *trap* interprets the extra 15 bits of information in the normal *src1, src2, dest* format whereby the registers are interpreted in either hardware or software.

2.1.4 Floating Point Instructions

Currently, the SDSP architecture does not explicitly define floating point instructions, since it is not supported by hardware. It can, however, be supported by system software via trap instructions. The trap instructions must use the existing integer operations to emulate the corresponding floating point operation. Perhaps, in the future, the instruction set will be expanded to include floating point operations, using the unused reserved class.

2.2 Architectural Organization

The architecture organization of the SDSP is shown in Figure 2.2. It is divided into three basic units:

Instruction Unit The Instruction Unit is composed of the Program Counter and branch predictor. Overall, it is responsible for flow control.

Scheduling Unit The Scheduling Unit consists of the instruction decoder, register file, reorder buffer, and instruction window. It is responsible for decoding instructions, deciding which instructions to issue to each functional unit, and maintaining the state of the machine.

Execution Unit The Execution Unit consists of functional units that can accept operands and return a result value. It also is responsible for communicating with the data cache. The SDSP uses four arithmetic logic units, a two-stage pipelined multiplier, a control transfer unit, a load unit, and a store buffer unit.

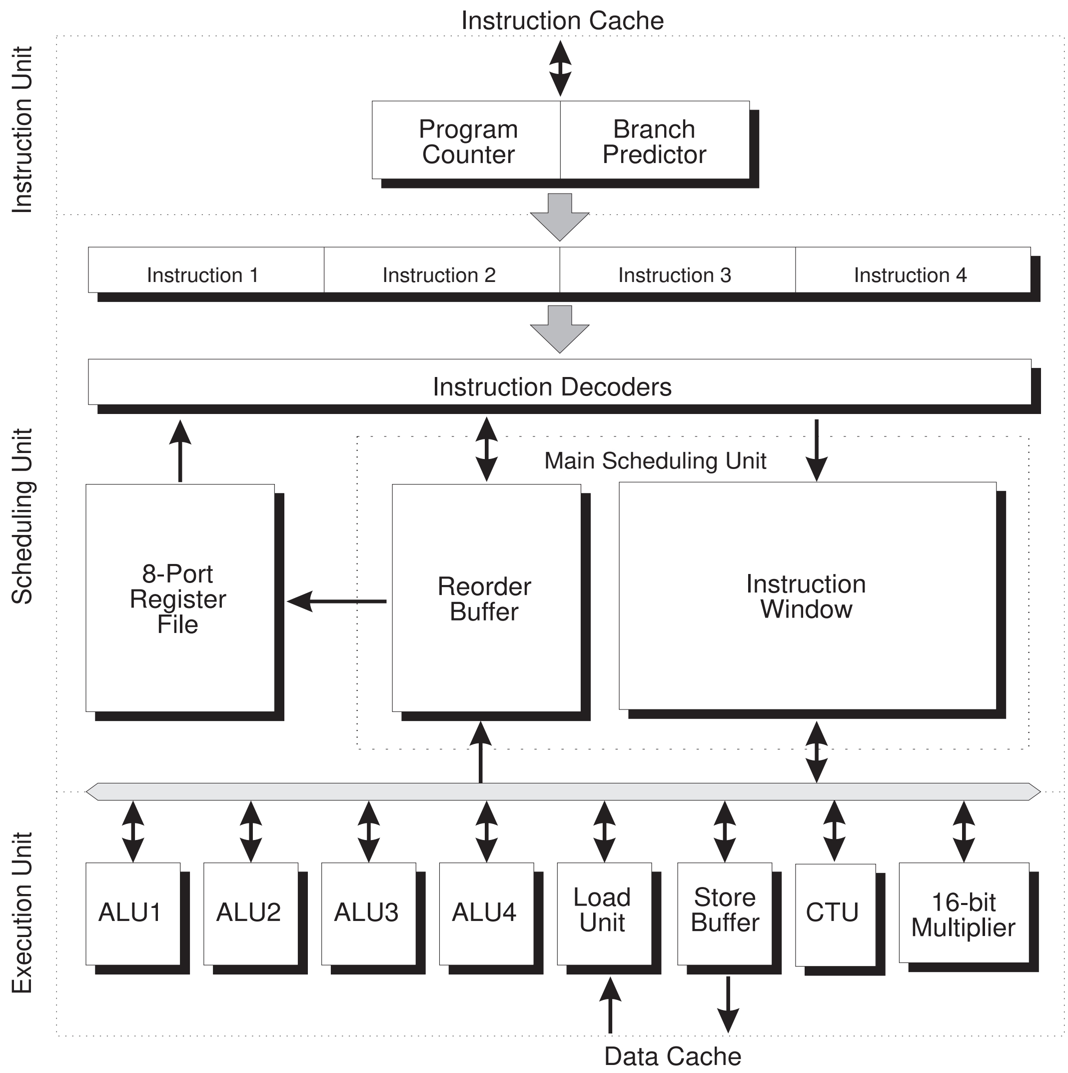

Figure 2.2: SDSP Architectural Organization

2.2.1 Pipeline Structure

The pipeline stages for the SDSP are the following:

IF Instruction Fetch block from instruction cache.

ID Instruction Decode and schedule onto functional units.

EX EXecute.

WB Write Back results to reorder buffer.

RC Result Commit results from reorder buffer to register file.

The structure of the pipeline is shown in Figure 2.3. Each pipeline stage is one cycle long.

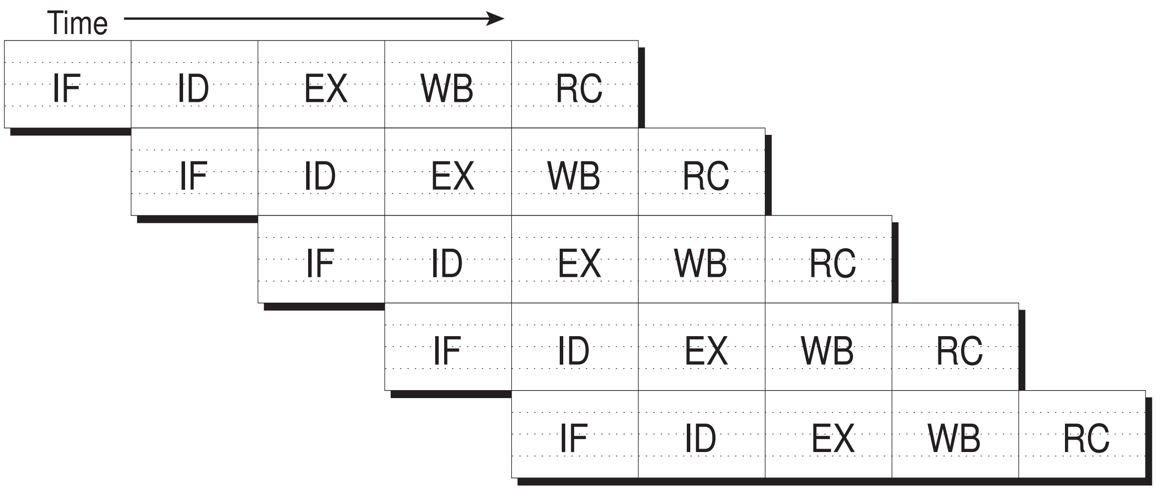

Figure 2.3: Pipeline Structure

2.2.2 Instruction Unit

The Instruction Unit maintains the control flow of the processor. The basic components of the Instruction Unit are the Program Counter and branch predictor.

Program Counter

The Program Counter (PC) handles the control flow of the processor. The memory address used by the Program Counter is organized in the format shown in Figure 2.4. The instruction block address ($A_{31} - A_{04}$) determines which block of four instructions to fetch. The instruction word address ($A_{03} - A_{02}$) is used to identify the *starting* instruction within a fetch block. Finally, the byte address ($A_{01} - A_{00}$) is used to identify individual bytes within a word in software, but is ignored completely in hardware.

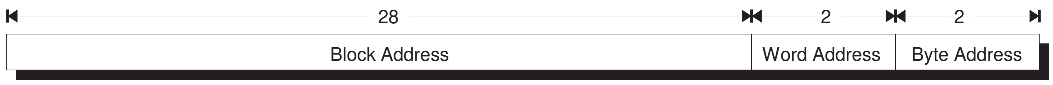

Figure 2.4: Memory Address Configuration

Branch Predictor

SDSP uses a superscalar branch predictor to support speculative execution [7]. The predictor is a modified branch target buffer [10], and is capable of performing multiple branch predictions per cycle. It uses an up-down saturating counter to evaluate branch prediction for each branch instruction encountered. Figure 2.5 is a block diagram of SDSP's branch target buffer. Each entry in the prediction buffer is indexed by the instruction *block* address which contains prediction information for up to four branch instructions in the block. The predictor evaluates all branch fields in the entry simultaneously to determine the first branch that is predicted "taken", if any. The multiple prediction feature eliminates the need to re-fetch the same instruction block if a branch is predicted "not-taken". Therefore, up to four predictions are accomplished in one cycle. When the Scheduling Unit commits a block of results to the register file, branch predictor statistics are updated based on outcome of branch predictions in that block.

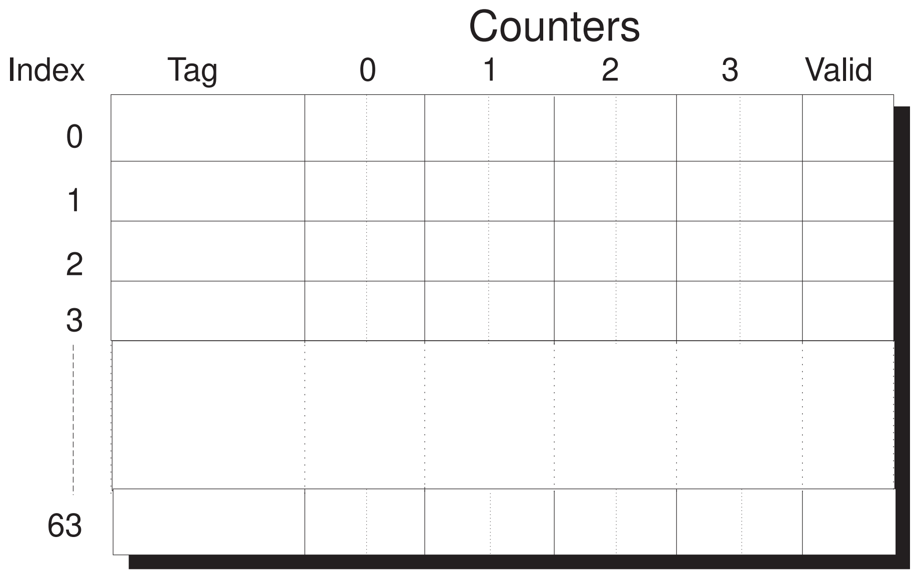

Figure 2.5: Block Diagram of SDSP's Branch Target Buffer

2.2.3 Scheduling Unit

The Scheduling Unit (SU) is responsible for decoding and issuing instructions to the Execution Unit which are received from the Instruction Unit. It is also responsible for maintaining the proper state of the machine. These tasks are accomplished using an instruction window, reorder buffer, and register file [8, 11]. A block diagram of the main Scheduling Unit is shown in Figure 2.6.

Dynamic out-of-order scheduling is achieved using a reorder buffer and a central instruction window, which behave as a single FIFO (First in, First Out) unit. The *SU depth* is defined to be the total number of entries divided by the decode size (four for the SDSP). It is equivalent to the number of shift cycles it takes for the instruction to be committed to the register file and leave the Scheduling Unit.

A block of four instructions received from the Instruction Unit are inserted at the top of the SU. If possible, the SU shifts a block of four instructions each cycle. The completed block of instructions at the bottom of the SU are committed to the register file and branch predictor. A *SU stall* occurs when the SU cannot shift because the reorder buffer is waiting for a result or the instruction window has not issued an instruction in the last block. Ready-to-run instructions, however, continue to be issued from the SU regardless of the type of stall.

Register File

The register file is able to handle a worst-case decode of eight source operands (two from each of four instructions) and accept four result commits from the reorder buffer. Consequently, an 8-port read and 4-port write RAM cell is used to accomplish this task.

Reorder Buffer

The reorder buffer is composed of a destination register field, data field, ready field, destination tag field, current field, and valid field. The name, quantity, type of

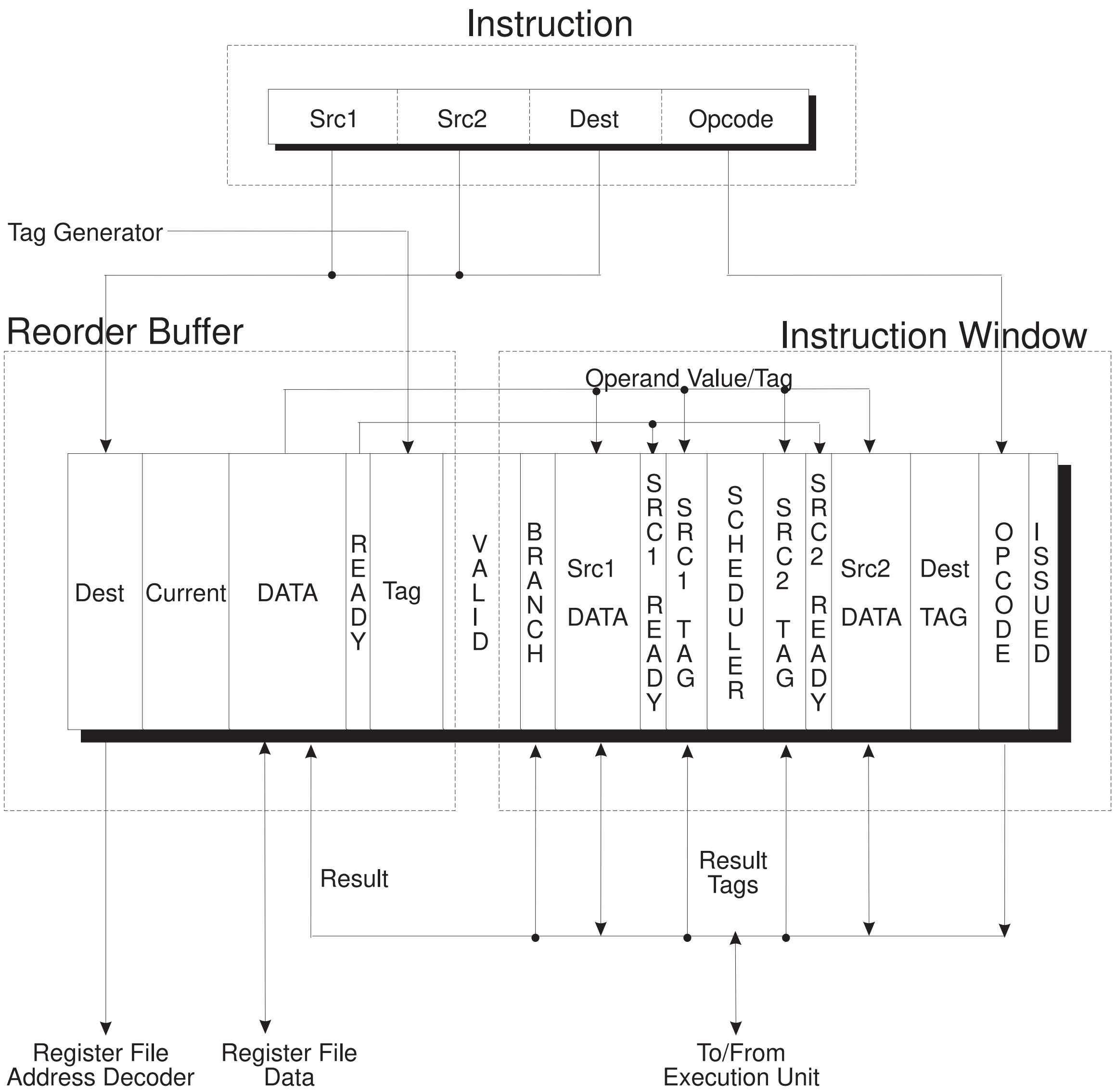

Figure 2.6: Block Diagram of Reorder Buffer and Instruction Window

cell (RO = Read Only, RW = Read/Write, CAM = Content Addressable Memory, RAM = Random Access Memory), and description of each field is given in Table 2.4.

Name	Qty	Type of Cell	Description
DEST	5	8-port RO CAM	Holds destination register and compares source registers.
DATA	32	8-port RW RAM	Holds 32-bit data value or 5-bit tag value.
READY	1	8-port RW RAM	Indicates if data or tag is in DATA field.
TAG	5	8-port RO CAM	Holds destination tag value and compares result write tags.
CURRENT	5	special lookup	Custom cell used in searching for most current destination register.
VALID	1	8-port RW RAM	Indicates if entry is valid.

Table 2.4: Reorder Buffer Fields

Each cycle the reorder buffer writes the results from the end of the FIFO. If a result is not ready and is required, then it will stall the entire SU. Otherwise, it shifts by four and creates new entries using the *dest* field from the new instructions. To accomplish decoding, the *src1* and *src2* fields from new instructions are used to search for matching *dest* entries in the reorder buffer. A special lookup cell is used to search for the most current *dest* match in constant time. Then, the value (if ready) or tag is read from the appropriate data cells and sent to the instruction window. To write results received from the Execution Unit, the result tag values must have been previously received and compared against all tag values in the reorder buffer. Then the result may be written to the data cells in the matching entry.

Instruction Window

The instruction window is composed of two source operand fields, two source ready fields, destination tag field, branch prediction field, scheduling field, opcode, and valid field. The name, quantity, type of cell, and description of each field is given in Table 2.5.

Name	Qty	Type of Cell	Description
BR. PRED.	1	8-port RW RAM	Contains *expected* value of LSB in *src1*.
SRC1 DATA	32	8-port RW RAM	Holds source 1 value.
SRC1 READY	1	8-port RW RAM	Indicates if source 1 is ready.
SRC1 TAG	5	8-port RO CAM	Holds source 1 tag value and compares result write tags.
SRC2 DATA	32	8-port RW RAM	Holds source 2 value.
SRC2 READY	1	8-port RW RAM	Indicates if source 2 is ready.
SRC2 TAG	5	8-port RO CAM	Holds source 2 tag value and compares result write tags.
DEST TAG	5	8-port RO RAM	Holds destination tag value.
OPCODE	4	8-port RO RAM	Holds opcode from instruction.
SCHED	5	special lookup	Custom cell used in searching and scheduling ready-to-run instructions.
ISSUED	1	8-port RW RAM	Indicates if entry has been issued.
VALID	1	8-port RW RAM	Indicates if entry is valid.

Table 2.5: Instruction Window Fields

The instruction window is primarily responsible for issuing operations to the functional units. The instruction window knows which functional units are available each cycle for scheduling. Using a special lookup cell, the instruction window searches up from the oldest entry to the youngest entry for the first ready-to-run instruction (that is, it has values for all required operands) for each available functional unit. It determines the tag values of results that will be received next cycle and compares them to the *src1* and *src2 tag* fields. When the results are returned, the values are stored in all appropriate *src1* and *src2 data* fields. Complete bypassing of results to functional units is used. Otherwise, the latency of each operation is effectively increased by one cycle. Shifting is allowed if all entries in the outgoing block have issued their instructions.

Bad Branches and Valid Bit

The instruction window and reorder buffer share a common valid bit for each entry in the SU. Therefore, when the valid bit is reset, both the entry in the reorder buffer and the entry in the instruction window are invalid.

A shared valid bit is useful when applied to speculative execution. A branch may be taken if the least significant bit of a single source operand is either set or reset. When a result from a previous instruction is written to the source operand of a branch instruction inside the instruction window, its least significant bit is compared to the expected result from branch prediction. If they are identical, no additional action is taken. If they differ, it means that the branch instruction was incorrectly predicted, and all entries that came after it in the SU must be invalidated. Because of the FIFO nature and physical layout of the SU, this invalidation process is achieved in hardware in constant time. Hence, branch instructions are executed *inside* the

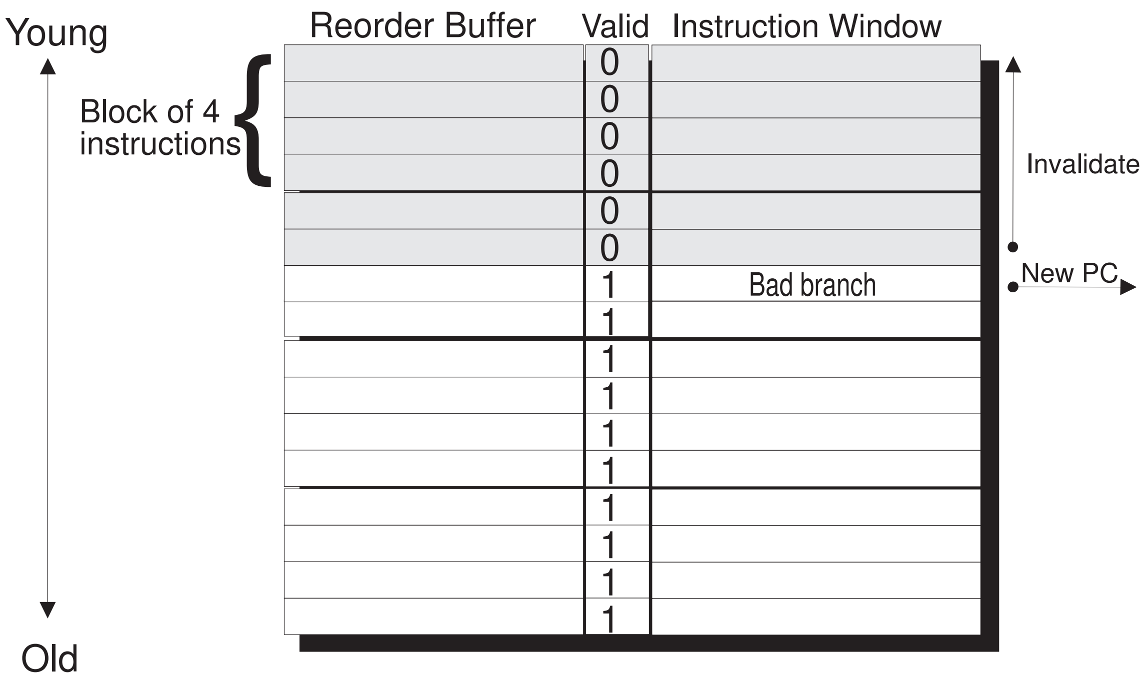

Figure 2.7: Bad Branch in a SU

instruction window. Only if an incorrectly predicted branch occurs will the Scheduling Unit notify the Instruction Unit of a new Program Counter.

For example, Figure 2.7 shows a SU of depth 4 and decode size of 4 where a bad branch has been detected after one cycle in the third instruction of a four instruction block. As a result, all instructions/entries which came after it (above it, with respect to layout), are invalidated and a new PC is sent to the control transfer unit.

Scheduling Algorithm

Since dynamic scheduling is complicated, choosing an appropriate scheduling algorithm from a hardware point of view is crucial for speed and design simplicity. The SDSP issues instructions using the "oldest first" scheduling algorithm. Searching instructions in hardware is simplified due to the layout of the SU, since the oldest instructions are at the bottom of the SU and the most recent instructions are at the top of the SU. An instruction is considered for potential scheduling if its operands are ready, it will receive its remaining operand(s) from a functional unit at the end of the current cycle, or it is currently being decoded by the reorder buffer. Proper ordering of loads and stores is accomplished by in-order issuing of stores but allowing out-of-order issuing of loads with respect to stores. In addition, a store may not be committed to memory until previous branches are resolved.

The result tag of completing operations must be known before scheduling. For functional units that are guaranteed to have their results ready at the end of the cycle, this presents no problem, but some functional units (such as load) may not have their results ready. Therefore, the functional unit assigned to an instruction that incorrectly assumed availability of its operands goes unused the following cycle, and the instruction will have to be rescheduled. Since this happens infrequently, this assumption is reasonable.

Issue Rate

The SU may issue up to eight different instructions per cycle (4 ALU, 1 multiply, 1 load, 1 store, and 1 control transfer), twice the decode size of four. The SDSP implementation accomplished this without increasing its size. In fact, by connecting each port directly to a functional unit, this decreased the scheduling complexity in hardware and removed multiplexors that would have to be used in a four instruction issue bus scheme. A four instruction issue may be used; this is an implementation decision.

Result Write Policy

The SU can receive up to four result values and corresponding destination tags each cycle from the Execution Unit. This creates a problem since six results may be ready in a particular cycle (4 ALU, 1 multiply, and 1 load). This is resolved by giving priority to load and multiply results over ALU results. The load result would override the last scheduled ALU unit (the youngest ALU instruction of the four), and the multiply result would override the third scheduled ALU unit. Consequently, two potential ALU units may not be able to return their result back to the Scheduling Unit, so they cannot be scheduled the following cycle. If the probability of this occurring is low enough, the performance impact is minimal.

2.2.4 Execution Unit

The execution unit consists of four integer Arithmetic and Logic Units (ALU), one 16-bit integer multiplier, one load unit, one store unit, and one control transfer unit. Each functional unit takes as input the required operands, control signals, and destination register tag. A functional unit may or may not return a result value.

Arithmetic Logic Unit

The arithmetic and logic unit requires two 32-bit source operands and returns one 32-bit result. It is guaranteed to complete within one cycle. It performs addition, subtraction, comparison, simple logic functions, and shifting. Comparisons may be signed or unsigned. The shifting is accomplished using a barrel shifter which can shift any amount in both directions. Four ALUs are used in the SDSP's Execution Unit.

Multiplier

The primary distinction between a general purpose microprocessor and a digital signal processor is the inclusion of an integer multiplier, since multiplications are commonly found in digital signal processing applications. The SDSP multiplier requires two 16-bit signed source operands in 2's complement format. It returns one 32-bit signed result value. Currently, the SDSP's multiplier implementation is pipelined with a 2 cycle latency. A one cycle multiplier is desirable and should be used, if possible.

Load Unit

The load unit is responsible for handling read operations from the data cache and the data memory. The load unit accepts a 32-bit source operand which contains the address of the data. It will return a 32-bit data result the next cycle upon a data cache hit or if it finds the data from a matching address in the store buffer. If the data is not in the cache, it is loaded externally before the result can be returned. Hence, load operations are not guaranteed to complete within one cycle.

Store Buffer Unit

The store buffer receives two operands: a 32-bit data address and a 32-bit data value. The value and address are stored into a buffer and are flushed to memory when the data bus is not in contention with a load operation. It does not return a result. It allows load bypassing of stores since the addresses are stored in a special content addressable memory cell. A new store instruction may be issued in the following cycle if the store buffer is not full. The number of entries in the buffer is implementation dependent; four is recommended. Therefore, using a store buffer allows both a load and store instruction to be issued in a single cycle using a single-port data cache.

Control Transfer Unit

The control transfer unit receives one source operand and does not return a result. The source operand is a new 32-bit PC address which replaces the current Program Counter.

Chapter 3

Tools and Simulation Methods

3.1 Compiler, Assembler, and Linker

A C compiler was ported from GNU CC version 1.42. The machine description and target machine definition files are similar to the MIPS format. Information on how to port the compiler is given in Appendix A. Table 3.1 lists the 32 integer registers and their assigned purpose for the compiler and system.

The assembler was ported from the GNU GAS assembler version 1.38. It uses the a.out object module format. Initial porting work was done by Mark Blair [2]. A special mnemonic, *la*, is defined to load a 32 bit immediate address using an *addi* followed by a *lui* instruction. Appendix B shows how to port the assembler.

The linker was ported from the GNU 'ld' linker which was modified by Donn Seeley at UUNET Technologies, Inc. The steps to port the linker are listed in Appendix C. Special modification was done so that loading a 32-bit immediate address using two instructions was adjusted properly when it crossed a 64 KB bank boundary.

Register(s)	Usage
0	Value is always 0.
1	Reserved for compiler and assembler.
2	Holds integer function results.
3	Reserved.
4-7	First 4 integers passed for function calls; values not preserved across function calls.
8-15	Temporary registers; values not preserved during calls.
16-23	Saved registers; values preserved during calls.
24-25	Temporary registers - not preserved.
26-28	Reserved.
29	Stack pointer.
30	Saved register; value preserved during call.
31	Return address.

Table 3.1: Register Usage

3.2 Simulator

After compiling, assembling, and linking the original C source code, the object code is loaded and executed in a instruction level simulator. It simulates the actual hardware representation as accurately as possible. It performs a cycle by cycle simulation and gathers statistics about its execution. The simulator was written entirely in C, with parts specifically customized for the SunOS host operating system. Appendix D tells how to obtain the source code.

Optionally, the simulator has an X Window user environment in which one can get a display of the state of the machine at any particular cycle (step through cycles). Figure 3.1 shows a sample snapshot of a simulated program in the X Window environment. The figure shows an instruction window with a decode size of four and a depth of eight. The top section is the most recently fetched block while the bottom section is the oldest fetched block of instructions. Each entry lists the type of instruction (ALU, MUL, LOAD, STORE, CNTL, FLPT), the Program Counter address, the instruction mnemonic, its source and destination register numbers, the source and destination tag values that the reorder buffer uses, its source and destination values (?, if not determined yet), and whether or not it has been issued. Each entry can be in four different states which are color coded on the display. The instruction can be waiting for operands (red/normal), ready to execute but not issued (blue/medium), issued but not completed (green/light), or completed (black/bold). To be able to "see" how the instruction window, reorder buffer, and register file change during execution has been useful.

Description	Range	Default
Instructions fetched per cycle (decode size)	1,2,4	4
Scheduling Unit depth	1-16	8
Maximum result writes per cycle	1-8	4
Maximum instructions issued per cycle	1-8	8
Line size (bytes)	4,8,16,32	16
Instruction cache simulation	Yes/No	No
Instruction cache miss penalty	1 - n	6
Data cache simulation	Yes/No	No
Data cache miss penalty	1 - n	6
Branch prediction	Yes/No	Yes
BTB branch predictor	Yes/No	Yes
Number of ALU units	1 - n	4
Number of multiplier units	1 - n	1
Number of load units	1 - n	1
Number of store units	1 - n	1

Table 3.2: Simulator Parameters

FU	PC	INST	R:SRC1	SRC2	DEST	T:S1	S2	DE	Val:SRC1	SRC2	DEST	ISS
STOR	58E20	stw	$ 0,	$ 2,	$ 0	0	4	5	0	????????	????????	N
ALU	58E24	addi	$ 4,	14,	$ 4	22	0	6	6F7F0	14	????????	N
ALU	58E28	addi	$ 0,	3,	$ 3	0	12	7	0	3	????????	N
STOR	58E2C	stw	$ 3,	$ 4,	$ 0	7	6	8	????????	????????	????????	N
STOR	58E10	stw	$ 0,	$ 2,	$ 0	0	0	1	0	????????	????????	N
ALU	58E14	addi	$ 4,	c,	$ 2	22	0	2	6F7F0	C	????????	Y
STOR	58E18	stw	$ 0,	$ 2,	$ 0	14	2	3	0	????????	????????	N
ALU	58E1C	addi	$ 4,	10,	$ 2	22	0	4	6F7F0	10	????????	Y
ALU	58E00	addi	$ 0,	8cd8,	$ 3	0	2	29	0	8CD8	8CD8	Y
ALU	58E04	lui	$ 3,	5,	$ 3	29	0	30	8CD8	5	58CD8	Y
STOR	58E08	stw	$ 3,	$ 2,	$ 0	30	28	31	58CD8	????????	????????	N
ALU	58E0C	addi	$ 4,	8,	$ 2	22	0	0	6F7F0	8	????????	Y
ALU	58DF0	addi	$ 0,	8db0,	$ 3	12	30	25	0	8DB0	8DB0	Y
ALU	58DF4	lui	$ 3,	5,	$ 3	25	0	26	8DB0	5	58DB0	Y
STOR	58DF8	stw	$ 3,	$ 4,	$ 0	26	22	27	58DB0	6F7F0	????????	N
ALU	58DFC	addi	$ 4,	4,	$ 2	22	0	28	6F7F0	4	????????	Y
STOR	58DEC	stw	$ 4,	5a98c		22	0	24	6F7F0	5A98C	????????	Y
CNTL	45810	calli	58dec			13	0	23	58DEC	58DEC	2	Y
ALU	45800	add	$ 6,	$ 0,	$20	25	0	19	2	0	2	Y
ALU	45804	add	$ 7,	$ 0,	$22	26	20	20	F7FFF6DC	0	F7FFF6DC	Y
ALU	45808	addi	$18,	4,	$16	17	0	21	6F8FC	4	6F900	Y
LOAD	4580C	ldw	$ 4,	$16,	$ 0	0	21	22	6F8FC	6F900	6F7F0	Y
STOR	457F0	stw	$16,	$ 1,	$ 0	12	14	15	6F8FC	6F7C8	6F8FC	Y
ALU	457F4	addi	$29,	34,	$30	28	0	16	6F7A4	34	6F7D8	Y
ALU	457F8	add	$ 4,	$ 0,	$18	23	17	17	6F8FC	0	6F8FC	Y
ALU	457FC	add	$ 5,	$ 0,	$23	24	0	18	0	0	0	Y

Figure 3.1: X Window Simulator - Instruction Window

The simulator is scalable and reconfigurable. The parameters which can change and their corresponding default values are listed in Table 3.2. A simulation proceeds first by loading the executable file. The text, data segments are relocated in memory. The relocated text and data segments may be dumped to a text file for viewing. BSS (run-time allocated data space) and stack space are allocated. The command line is passed as the first two parameters to main(). The simulator may load in a command file that specifies the parameters listed in Table 3.2. Redirection of standard input and standard output may be done through a file. The simulator may also produce an instruction trace. Most importantly, it produces a statistical output of the simulation run. The data collected is as followed:

- Total number of instructions executed.
- Total cycles used to execute program.
- Floating point cycles - total cycles spent emulating floating point instructions.
- Scheduling Unit stall cycles - number of cycles which the SU could not receive a new instruction block since SU could not shift out old instructions.
- Branch delay cycles - number of cycles during which incorrectly predicted blocks were fetched.
- Instruction cache delay cycles - number of cycles which the SU had to wait for a new fetch block.
- Data cache delay cycles - number of cycles which resulted in a SU stall due to a data cache miss.
- Total delay cycles in which no new blocks of instructions were fetched.

- Instruction cache miss ratio - number of instruction cache misses divided by number of instruction cache attempts.
- Data cache miss ratio - number of data cache misses divided by number of data cache attempts.
- Total write overflows - number of cycles in which the number of results exceeded result bus limitations causing a Functional Unit to delay a cycle.
- Issue count distribution - number of instances of 0 to a potential 8 instructions being issued in a given cycle.
- Write result count distribution - number of instances of 0 to 4 results being written in a given cycle.
- Branch count distribution - number of instances of 1 to 4 branches being executed in a given fetch block.
- Type count distribution - number of ALU, multiply, load, store, control, and floating point type instructions executed.
- Branch prediction ratio - number of branches successfully predicted divided by the number of prediction attempts.
- Average bad branch penalty - average number of incorrectly fetched blocks due to a mispredicted branch.
- Average Instruction Fetch count - average number of valid instructions per fetch attempt.
- Average number of integer instructions executed per cycle.

The simulator makes the following assumptions and considerations:

- Instruction Unit - Instructions from an unaligned block fetch are invalidated as well as instructions after a control transfer. No-ops are invalidated and not considered to be an instruction.

- Branch Prediction - If branch prediction is allowed, branches are predicted based on a 64 entry multiple prediction buffer or a simple "not taken" policy. When a branch is encountered, the simulator continues down its predicted path until it is determined to be incorrect.

- Scheduling Unit - Instructions may be issued out-of-order by using the "oldest first" scheduling algorithm. An instruction is issued if it is or will be ready to run, the corresponding functional unit resource is available, and the issue limit parameter is not exceeded.

- Execution Unit - Each functional unit may write back its result if it does not exceed the maximum result writes per cycle parameter. Priority is given to load, multiply, and ALU instructions, in that order. ALU operations execute in a single cycle. Multiply operations are pipelined with a two cycle latency. Load and store operations take a single cycle on a data cache hit (or perfect cache).

- Store buffer - The store buffer was not simulated. It is assumed a reasonable store buffer size would be used (say, 8 entries), so that the probability of stalls would be low and its effect negligible.

- Cache - The instruction cache is direct mapped. The data cache is write through direct mapped.

- Floating point operations are implemented as trap instructions. When a FP operation is encountered, the corresponding cycle latency required to emulate this function using integer operations is added, and the block is refetched, if necessary.

- Operating system calls are implemented as traps. When a trap to the OS is encountered, the simulator calls the corresponding OS function on the host system and then resumes simulation.

3.3 Benchmarks

The results in this thesis are from seventeen integer-intensive programs. All benchmarks were run until completion. Eight of the programs come from the Stanford suite of benchmarks. Eqntott comes from the SPEC '89 Integer benchmark suite. DCT is a custom benchmark which was extracted from the JPEG image compression standard since it is widely used in digital signal processing applications. It performs a discrete cosine transform on a 8x8 pixel image and then performs the inverse transform a total of 100 times. The MPEG, P64, and JPEG (compression only) were written by the Portable Video Research Group. The decompression for JPEG came from the Independent JPEG Group's software, release 4. A description of each program, the approximate number of instructions executed until completion, and the input file used is given in Table 3.3. Notice that execution ranges from just a half a million to one and a half *billion* instructions.

The type of each program is an important consideration in selecting a suite of benchmarks. A program's type can be deduced by the distribution of the type of instructions executed. Figure 3.2 shows that distribution for each program. This suite exhibits a wide variety of distributions, which is good for overall comparison.

Program	Description	Input	Instructions Executed
dct	Discrete Cosine Transform	none	500,000
dhrystone	The Dhrystone synthetic benchmark	20,000 reps	9,000,000
eqntott	Truth-table generator	int_pri_3.eqn	2,800,000
bubble	The Stanford bubblesort program	none	1,500,000
intmm	The Stanford integer matrix multiply	none	1,000,000
jpeg.c	JPEG compression	nonint.jpg	10,400,000
jpeg.d	JPEG decompression	testimg.jpg	6,000,000
mpeg.c	MPEG compression	short	1,509,000,000
mpeg.d	MPEG decompression	short.mpg	130,000,000
p64.c	P64 compression	short	672,000,000
p64.d	P64 decompression	short.p64	100,000,000
perm	The Stanford permutation program	none	2,000,000
puzzle	The Stanford puzzle program	none	8,500,000
queens	The Stanford eight-queens program	none	1,300,000
quicksort	The Stanford quicksort program	none	1,000,000
towers	The Stanford Towers of Hanoi	none	2,400,000
tree	The Stanford treesort program	none	2,000,000

Table 3.3: Benchmark Programs

Some programs are ALU intensive, others need multiplications, and some rely on loads and stores. The overall instruction type distribution over all the programs is shown in Figure 3.3. Approximately two thirds of the instructions executed are ALU, one quarter load and store, and one eighth control transfer.

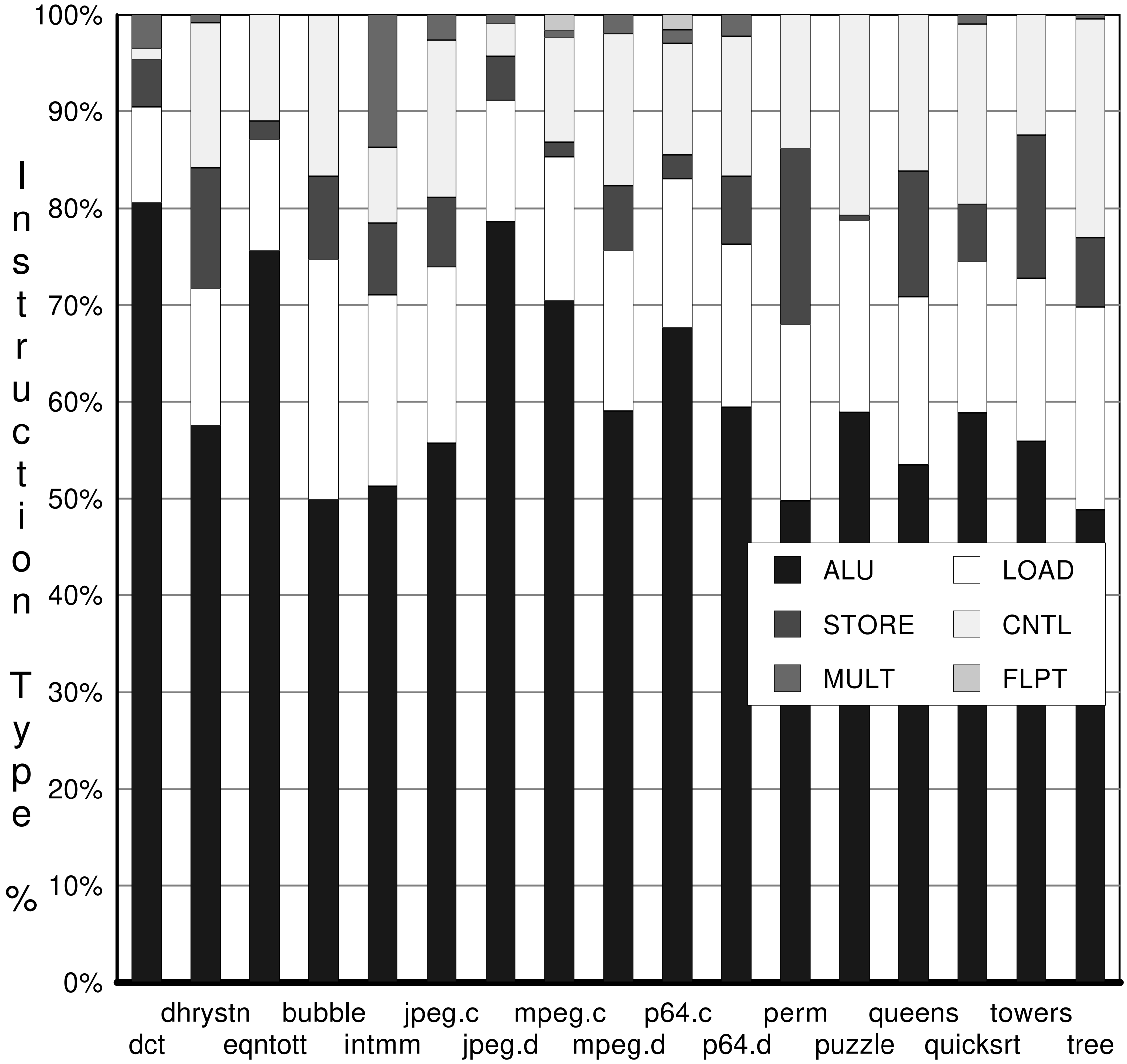

Figure 3.2: Instruction Type Distribution

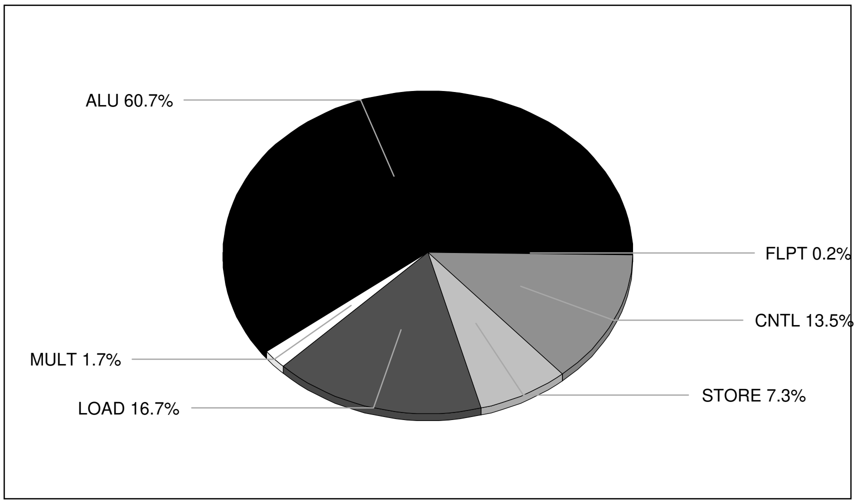

Figure 3.3: Overall Instruction Type Distribution

3.4 Default Configuration

Unless otherwise noted, the base configuration of the simulator uses the default as described in Table 3.2: four ALU units, one multiplier, one load unit, one store unit, 8 instruction issue limit, 4 result write limit, perfect cache, and a 64 entry multiple branch prediction buffer.

Chapter 4

Results

This chapter presents performance issues and data related to the Instruction Unit, the Scheduling Unit, and the Execution Unit. The last section considers overall performance.

4.1 Instruction Unit

The Instruction Unit is a key area in considering performance issues. It must continuously deliver blocks of instructions, otherwise all other performance enhancements will not be of any benefit. In addition, if instructions fetched are not valid, fetch efficiency will be reduced. Lastly, branches cause control flow hazards and must be dealt with in an appropriate manner using branch predictions.

4.1.1 Instruction Fetch Efficiency

The SDSP processor fetches a block at a time, which contains four instructions. However, all four instructions may not be valid because the first instruction is not at the first decode position or a control transfer was not at the last decode position.

The Average Instruction Fetch (AIF) count places an upper bound on the speedup of a program, since the processor cannot execute more instructions than it actually decodes. Figure 4.1 shows the Average Instruction Fetch count for each program for a decode size of 2 and a decode size of 4.

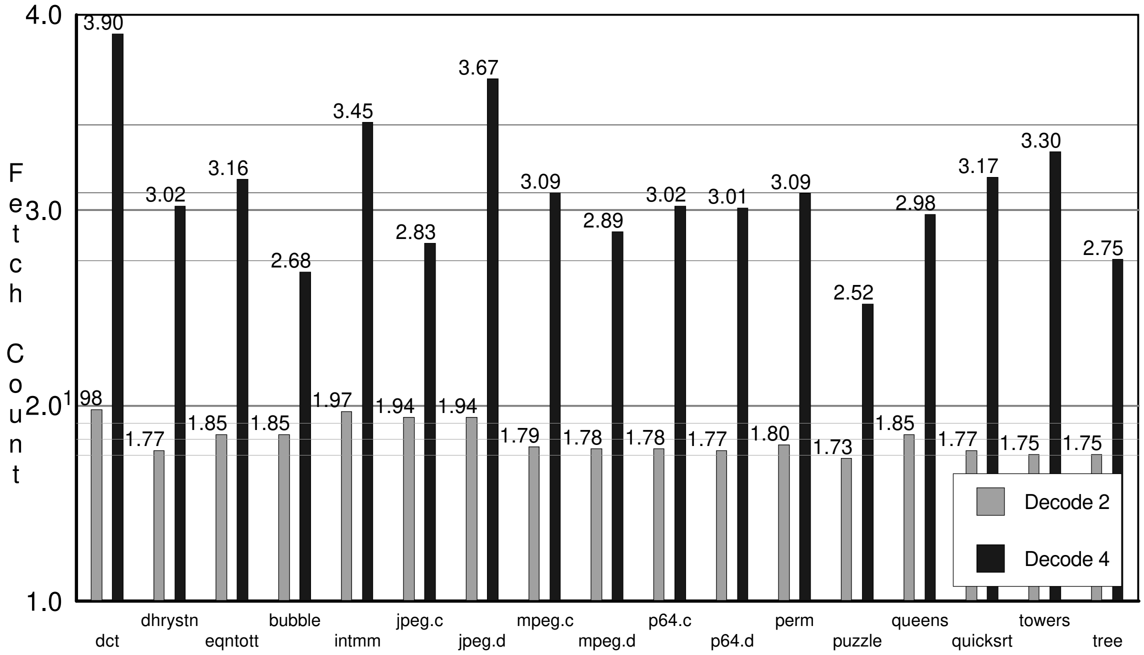

Figure 4.1: Average Instruction Fetch vs. Decode Size

Increasing the fetch bandwidth will increase the fetch efficiency. As expected, the AIF for a decode size of 4 is significantly larger than for a decode size of 2. Also shown in the figure are the average and standard deviation of for a decode size of 2 and 4. Although the decode size doubled, the AIF only increased by about 73%. The probability that a control transfer instruction's source or destination address is not on the last decoder position increases as the decode size of the block increases. Therefore, increasing fetch bandwidth has a marginal return to AIF.

Programs with less alignment usually reflect more control transfer instructions and lower AIF. In the case of DCT, the average number of instructions per fetch cycle is nearly perfect because instruction runs are long.

For a decode size of 4, Figure 4.2 displays the percentage of instances where 1, 2, 3, or 4 valid instructions were fetched and executed. As can be observed, over half the time 4 valid instructions were fetched. The other half is split up roughly evenly between 1, 2, and 3 fetches.

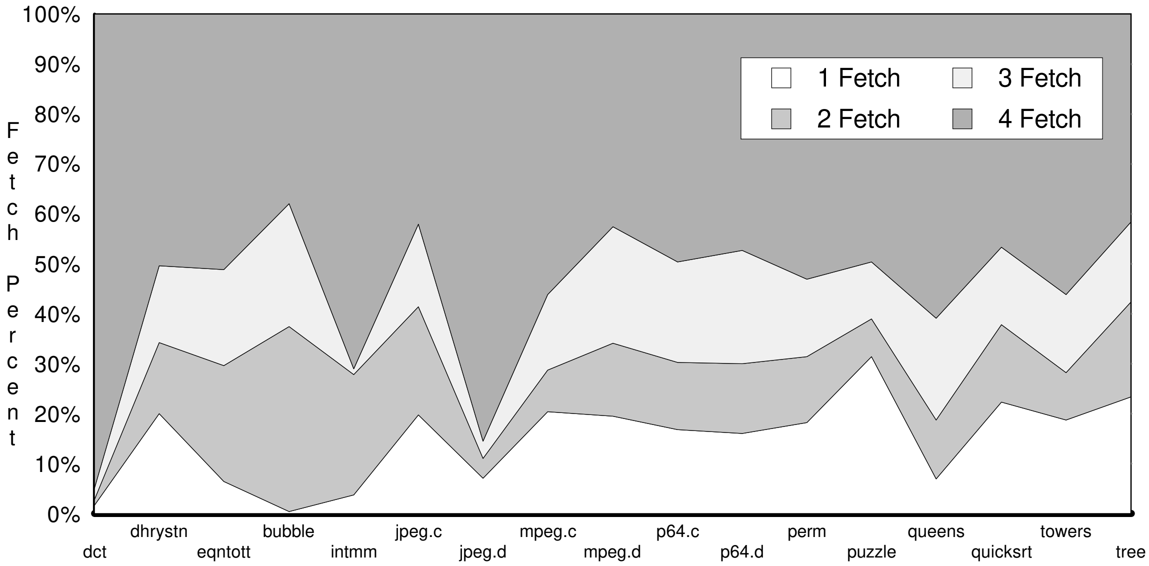

Figure 4.2: Instruction Fetch Distribution

4.1.2 Branch Prediction

Using a 64-entry prediction branch buffer, the programs were able to achieve an average branch prediction accuracy of 84%. On the other hand, if a simple "not taken" prediction policy were used, the programs achieved a relatively poor average branch prediction of 37%. The programs' accuracy for both schemes is shown in Figure 4.3.

One way to handle branches is to execute them in-order when the branch reaches the bottom of the SU. If a branch is mispredicted, the entire reorder buffer and instruction window are invalidated and execution resumes at the correct location. The penalty associated with a mispredicted branch would be the depth of the SU. This penalty is constant and grows linearly with respect to the size of the SU.

Alternatively, if the SU recovered from a mispredicted branch as soon as the outcome was known, substantial savings could be achieved. Table 4.1 shows the average bad branch cycle penalty for Scheduling Unit depth of 2, 4, 8, and 16 using this method with a 64-entry branch prediction buffer as described in Chapter 2.

The bad branch cycle penalty is the number of incorrectly fetched blocks, plus an additional cycle penalty for branches mispredicted taken that are not on the last decoder position, since it must refetch the same block upon recovery. On the other hand, cycles that stalled due to data dependencies in the Scheduling Unit are not considered part of the branch penalty. For instance, if after a mispredicted branch the SU stalls a cycle and resolves that branch, the Instruction Unit will resume fetching the correct block and will have never fetched an incorrect block. Hence, there is no penalty associated with that misprediction, since a correct prediction would have resulted in identical performance. This is how an average penalty of less than one cycle can be achieved.

4.1.3 Instruction Cache

Instruction fetching must be sustained, or it will be impossible to achieve a high Instructions Per Cycle rate. The instruction cache should supply a new block of instructions each cycle. Using a direct mapped instruction cache, Figure 4.4 shows

Program	2	4	8	16
dct	0.3	1.0	2.3	1.8
dhrystone	1.1	2.2	2.4	2.8
eqntott	1.0	2.5	4.2	4.5
bubble	1.0	3.0	4.0	4.0
intmm	1.0	0.1	0.5	1.0
jpeg.c	1.1	1.9	2.3	2.3
jpeg.d	0.9	2.5	4.2	5.7
mpeg.c	0.7	2.5	3.5	3.6
mpeg.d	0.7	2.5	3.4	3.6
p64.c	1.1	2.7	3.3	3.3
p64.d	0.9	1.7	3.0	3.3
perm	0.6	1.6	1.8	1.6
puzzle	0.8	2.6	3.2	3.2
queens	0.9	2.3	3.3	3.3
quicksort	0.9	2.4	2.9	2.9
towers	0.9	1.1	1.1	1.1
tree	1.0	2.8	2.8	3.0
Average	0.9	2.1	2.8	3.0

Table 4.1: Average Bad Branch Penalty Cycles vs. SU depth

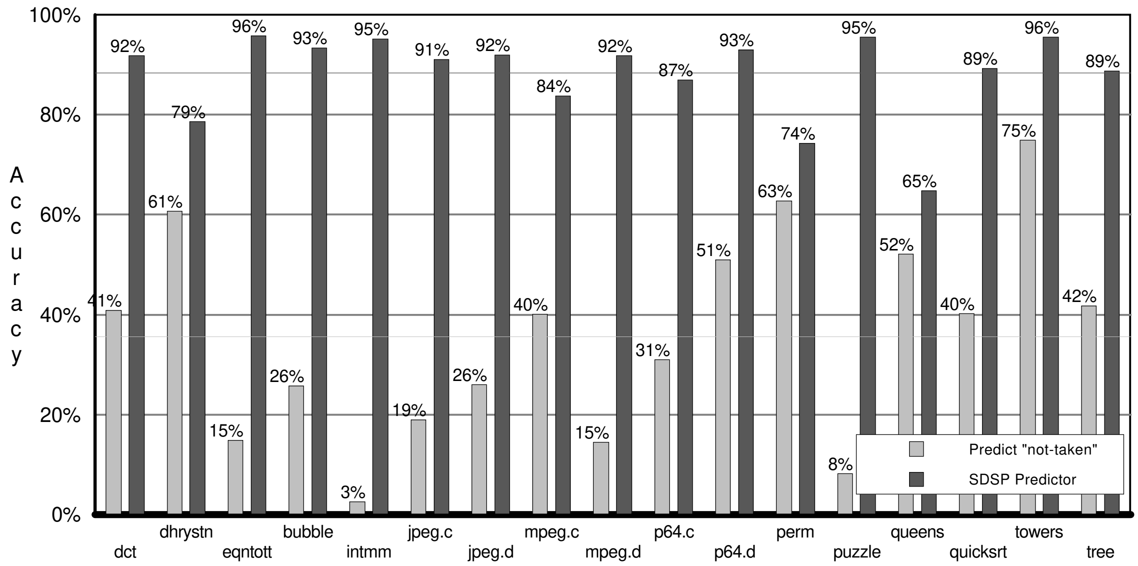

Figure 4.3: Branch Prediction Accuracy

the miss rate. Only the digital signal applications had a high enough miss rate to be considered. The rest, including the Stanford benchmarks, have a miss rate of less than 0.1%. An instruction cache of at least 4 KB is needed for a tolerable miss rate. Otherwise, performance loss due to instruction cache misses suffers dramatically at high miss rates.

4.2 Scheduling Unit

This section is concerned with the performance of the Scheduling Unit. The Scheduling Unit needs to be able to issue the majority of the ready-to-run instructions or else instructions will pile up. Furthermore, if the results are not able to write to the SU and activate additional instructions, this will reduce the ready-to-run pool of instructions. Also, if the Scheduling Unit depth is not large enough, it does not look-ahead enough to extract parallelism, and the SU will stall due to data dependencies.

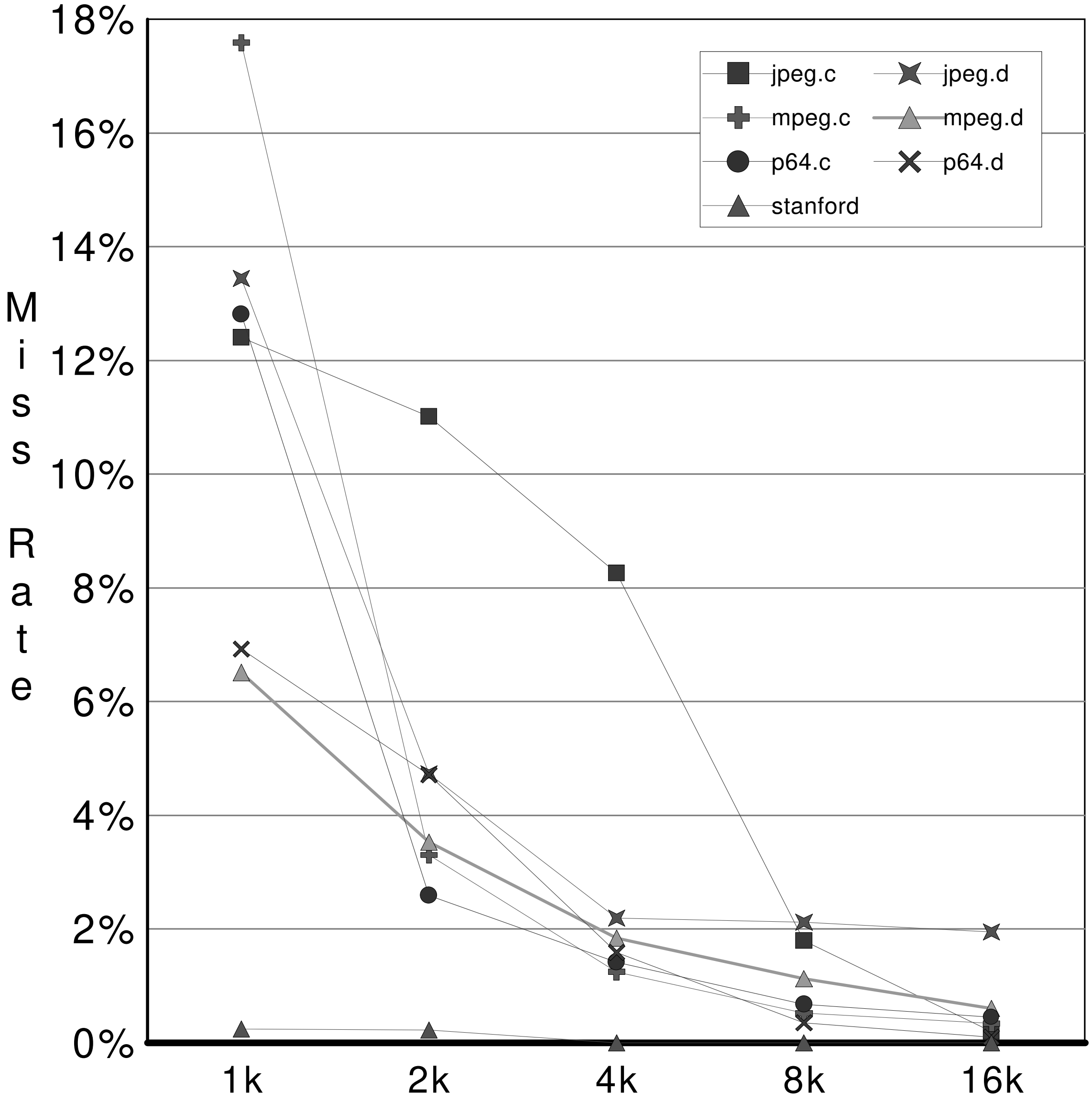

Figure 4.4: Instruction Cache Miss Ratio vs Size

4.2.1 SU Depth

Figure 4.5 shows the overall speedup using a SU depth of 2, 4, 8, and 16 for all the programs. The speedup difference between SU depth of 2 and 4 is substantial. The increase from a depth from 4 to 8 is significant in most cases. Comparing the speedup using SU depth of 8 to the average fetch count, as in Figure 4.1, shows that most programs have come close to reaching their maximum potential speedup. The difference is usually from mispredicted branch delays. If there is still a significant difference, increasing the SU depth to 16 will reduce this gap, as noticed with *dct, intmm, and jpeg.d*. Otherwise it will make no difference and can actually reduce speedup slightly as in the case with *bubble, perm, puzzle, quicksort, and treesort*. The loss in speedup is from a decrease in branch prediction accuracy. Branch prediction statistics are updated when a block of instructions is shifted out of the SU. Therefore, accuracy is sometimes lost when increasing the SU depth because this increases the latency required for update.

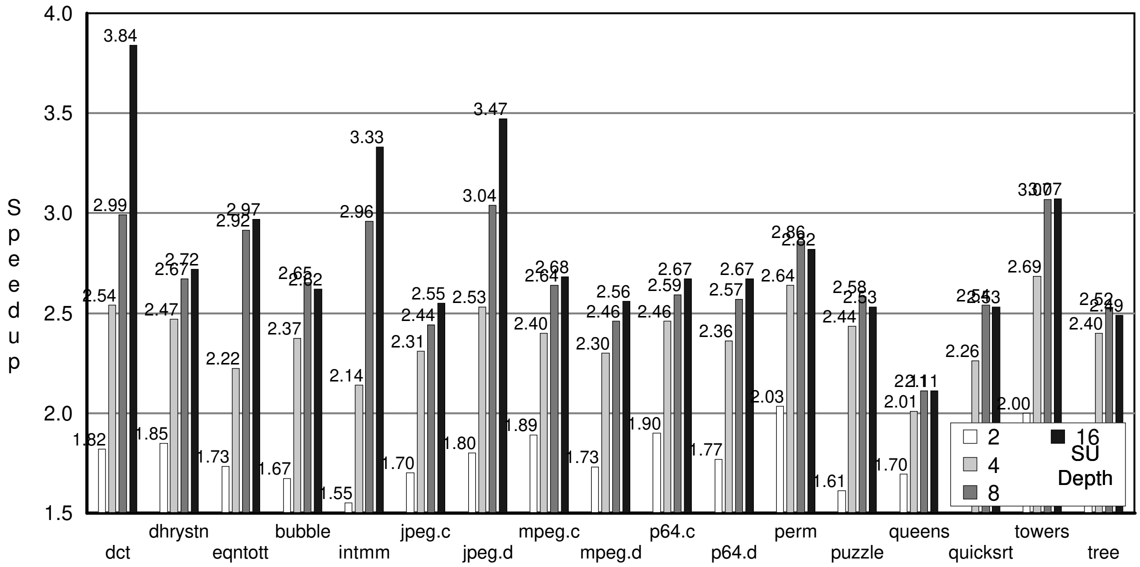

Figure 4.5: Speedup Improvement

4.2.2 Issue/Result Write Limit

The SDSP uses complete bypassing of results to functional units to avoid extra latency. This, however, is costly in hardware. Figure 4.6 shows the performance effect without bypassing compared to complete bypassing used in the base machine. As can be observed, bypassing has a two to three times savings in the number of stall cycles. This is a significant performance savings. Another way of looking at it is that the size of the SU needs to be over twice as large if bypassing is not used to get equivalent performance. For example, as shown in the figure, at a SU depth of 3 with bypassing would require a SU depth of 7 without bypassing for the same percentage of SU stalls. Hence, using bypassing is worthwhile since the extra space in layout is significantly less than a doubling of the size of the SU.

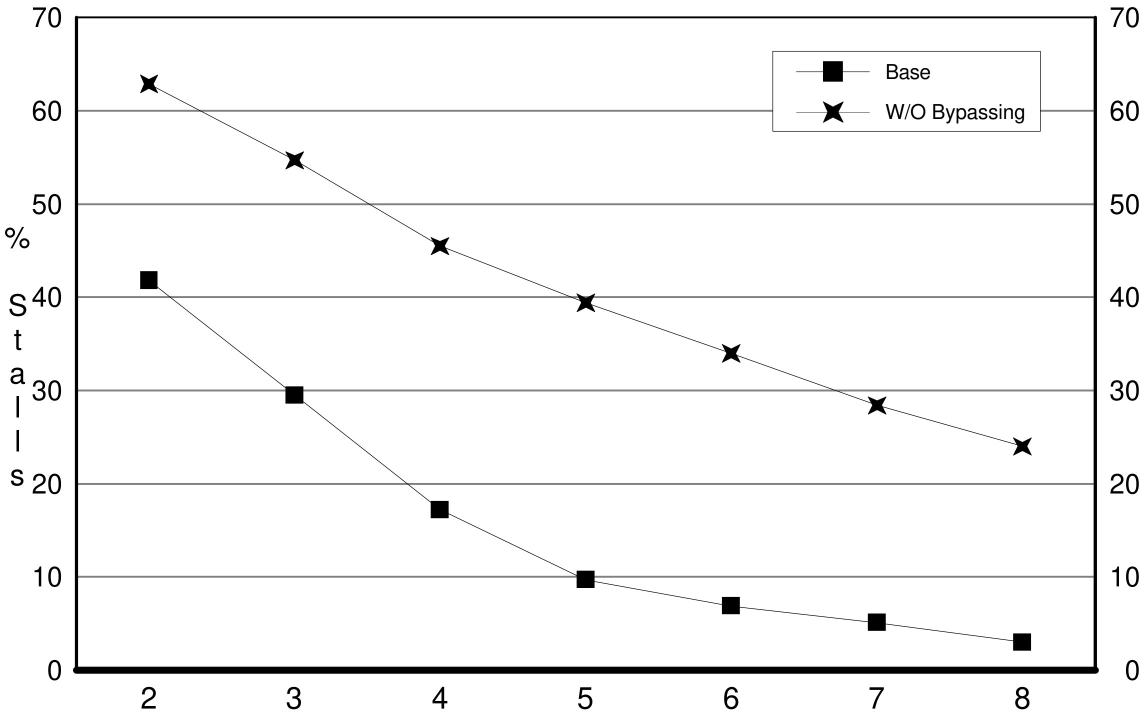

Figure 4.6: Percentage Stalls vs. SU Depth

Figure 4.7 shows the percentage difference between allowing 5 or 6 (identical outcome) result writes per cycle compared to the base case of 4. Since up to 8 instructions may be issued in a cycle, a maximum of 6 results may be ready to be written in a cycle. Allowing 5 result writes increased performance for many programs, especially for *dct, eqntott, jpeg.c, and mpeg.c* where over a 10% increase in speedup is observed. For the *bubble, intmm, p64.c, perm, queens, and quicksort* programs, however, absolutely no difference resulted from allowing 5 result writes. Programs with a sizable difference between AIF (Figure 4.1) and the base speedup (Figure 4.5, SU = 8) showed the greatest improvement. The case of 6 result write never occurred which implies 5 is the real maximum. Therefore, a 6 result write limit is unnecessary and wasteful.

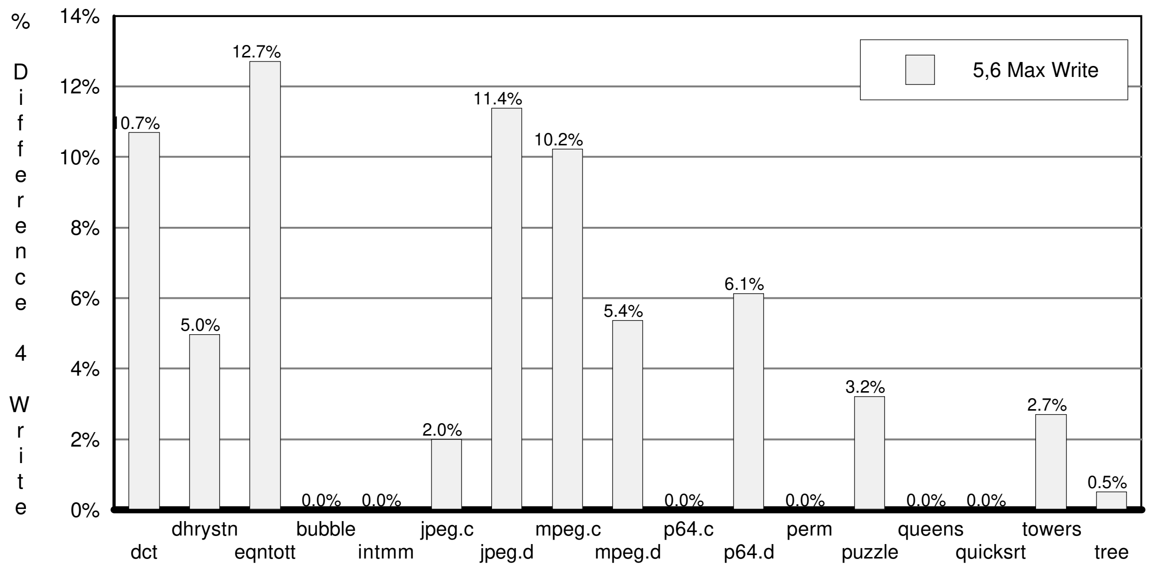

Figure 4.7: Performance for Write Result Increase

Figure 4.8 shows the issue count distribution. The bulk is evenly distributed between two, three, and four issue count, which would appear to average out to about three. About 15% of the time only one instruction can be issued. The low instruction issue count is partly due to fetch inefficiencies and data dependencies. On the other hand, approximately 10% of the time five or six instructions are issued. When five or six instructions are issued, the average is balanced from the cases where only one, two, or three instructions are issued so that the overall instructions issued per cycle is closer to four. Therefore, increasing the maximum issue count above the decode size does increase performance.

The instructions issued beyond four ALUs can be a load, store, multiply, or control transfer. Since store and control transfer operations do not have a result, increasing the maximum issue count without increasing the maximum write count can improve IPC. To receive benefit from when an extra load or multiply instruction is issued, one must increase the maximum result write per cycle as has been shown previously.

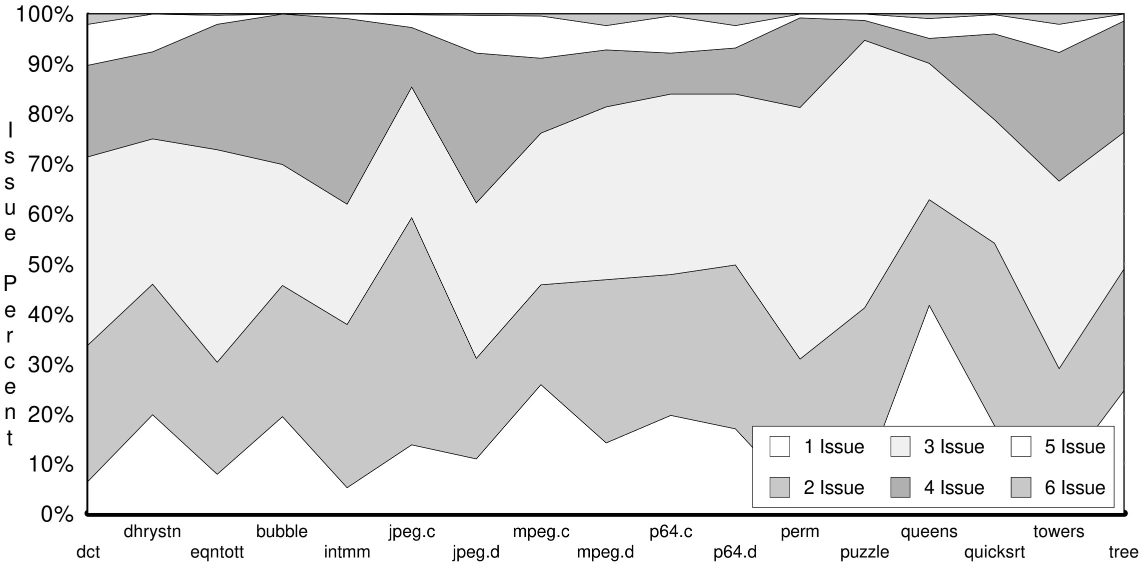

Figure 4.8: Issue Count Distribution

4.3 Execution Unit

Execution Unit issues in performance cannot be ignored. If a slow functional unit or not enough functional units of a particular type are used, a decline in performance will be observed. This section considers issues involving the data cache, ALU, multiplier, and load and store units.

4.3.1 ALU

Since only three instructions are executed per cycle on the average, does it make sense to use four ALUs instead of just three? On the other hand, since it is possible to issue more than four instructions a cycle, perhaps having five ALUs will avoid unnecessary stalls? To determine this, Figure 4.9 shows the speedup decrease percentage from using 2 or 3 ALUs compared to using the base case of 4 ALUs, keeping the result write limit constant at 4. Using only 2 ALUs decreases performance significantly for almost all cases since it cannot meet the resource needs of the SU. On the other hand, 3 ALUs performs almost identically to 4 ALUs, except for ALU-intensive programs such as *dct* and *jpeg.d*. The average speedup is less than 3.0 for most programs. Logically, 3 ALUs should suffice for these cases since a large SU can compensate for waiting ALU instructions. Using 4 ALUs and a 4 result write limit can create a write conflict with a load or multiply operation. Eliminating one ALU removes this conflict.

A more fair comparison is to isolate the ALU usage by allowing more result writes than ALUs to remove write conflicts with loads and multiplies. Figure 4.10 displays the performance difference of using 2, 3, or 5 ALUs compared to 4 ALUs.

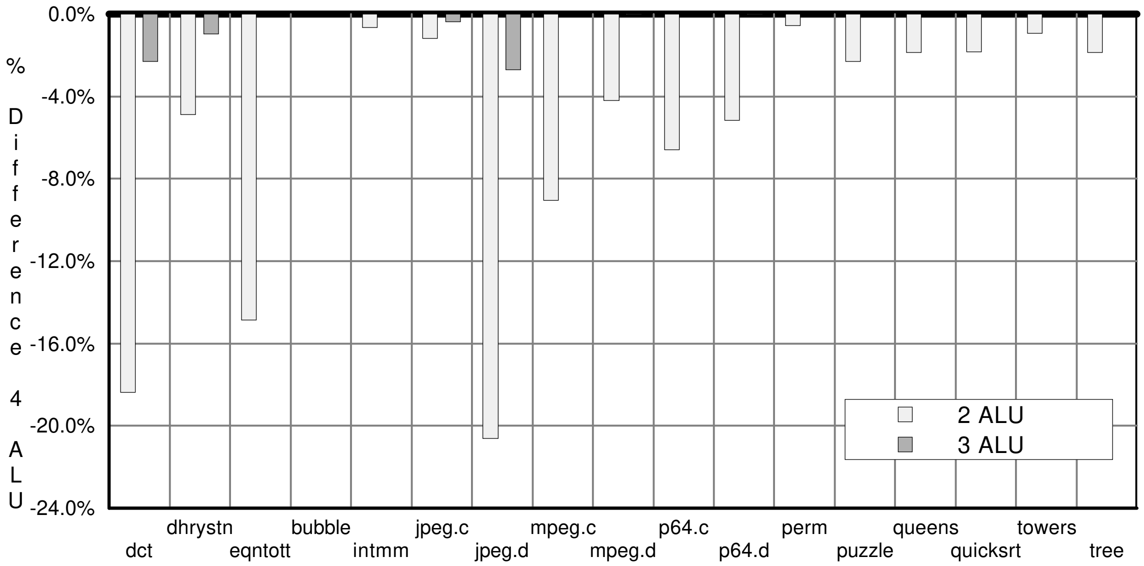

Figure 4.9: Performance Decrease from 4 ALUs, Write Limit Constant

Note, the basis for comparison uses 4 ALUs with 5 result writes which is different from the previous figure. In addition, the items compared have enough result write bandwidth so no write conflicts arise.

Programs that execute over 70% ALU (refer to Figure 3.2) operations and have a high average fetch count (refer to Figure 4.1) have the greatest change in performance when ALU resources change. Those programs (*dct, eqntott, and jpeg.d*) benefit significantly from using 5 ALUs. Now that the full potential of 4 ALUs is represented, 3 and especially 2 ALUs show dramatic performance decrease, although some programs' performance does not decline as much.

4.3.2 Multiplier

Since multiplication is used quite frequently in digital signal processing, one multiplier may not be enough to avoid delays. Figure 4.11 shows the performance

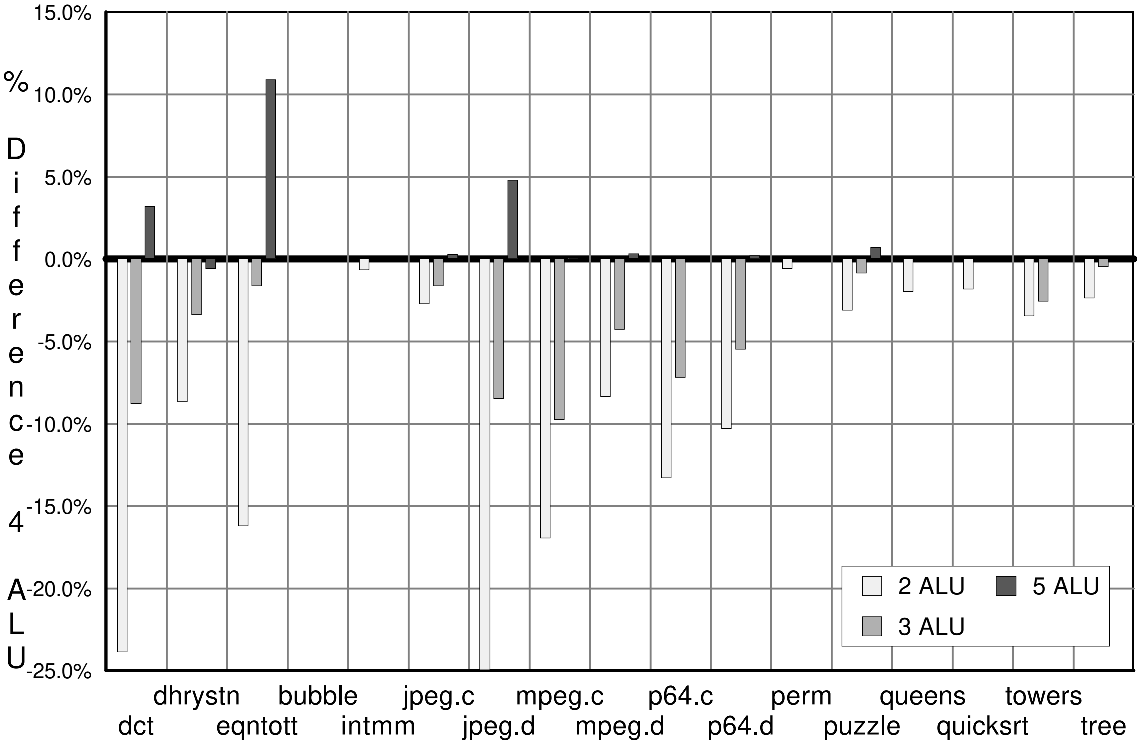

Figure 4.10: Performance Difference Compared to 4 ALUs, Write Limit Varied

increase using 2 multiplier units over 1 multiplier unit. A performance increase is observed only for the DSP applications (DCT, JPEG, MPEG, P64) and *intmm*. DCT showed the greatest increase followed by P64 and JPEG. This increase varied from 1% to 7% which translates to a 0.1 to 0.2 IPC increase. Overall, using 2 multipliers did not prove worthwhile.

4.3.3 Load/Store

Load operations can be critical because usually many following instructions depend on it. If a load has to wait because of a resource conflict, all its dependents will also have to wait. Currently, only one load operation may be done in a cycle. Store instructions may be issued once per cycle. That rate cannot be sustained since

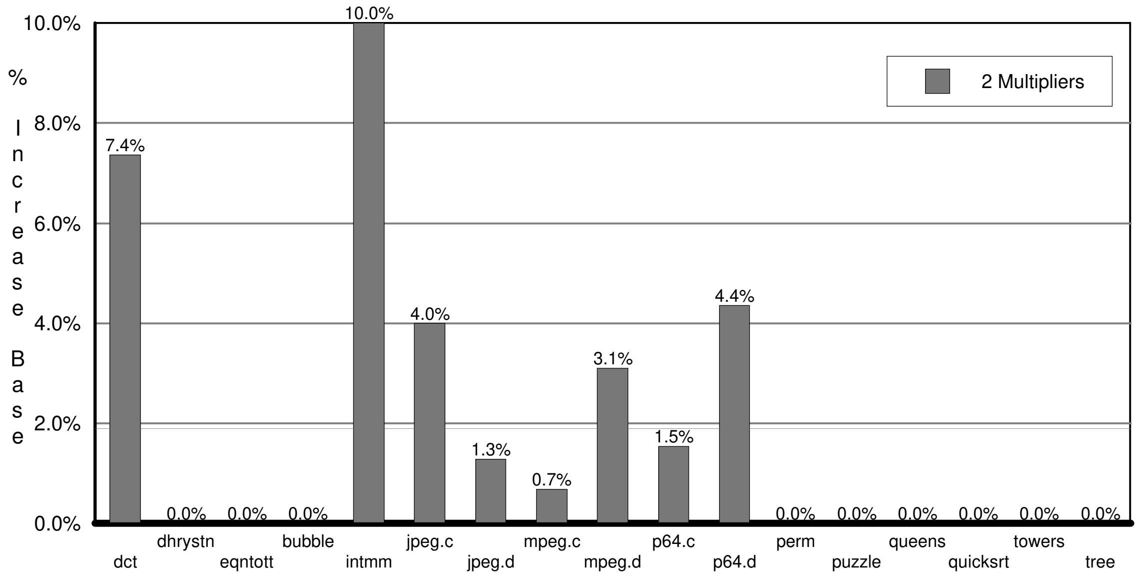

Figure 4.11: Performance Increase Using 2 Multipliers

both load and store instructions share the data cache and load instructions have priority. If, on the average, only one quarter of the instructions are load and store instructions, a single data port cache should suffice. Problems arise when two or more load or store instructions are ready to be issued at the same time.

Using a two-port data cache will expand the load/store bandwidth and can reduce stalls. Figure 4.12 compares the speedup of using two load and two store units versus one load and one store unit. A few cases which have many consecutive load or store operations such as *dct* and *queens* have a significant increase in performance, but for most cases the increase was less than one percent. It should be noted, since the store buffer overflows were ignored in the simulator, programs with both high load and store content would benefit more than is shown in the figure. Therefore, one will have to weigh the costs involved before implementing a two-port data cache.

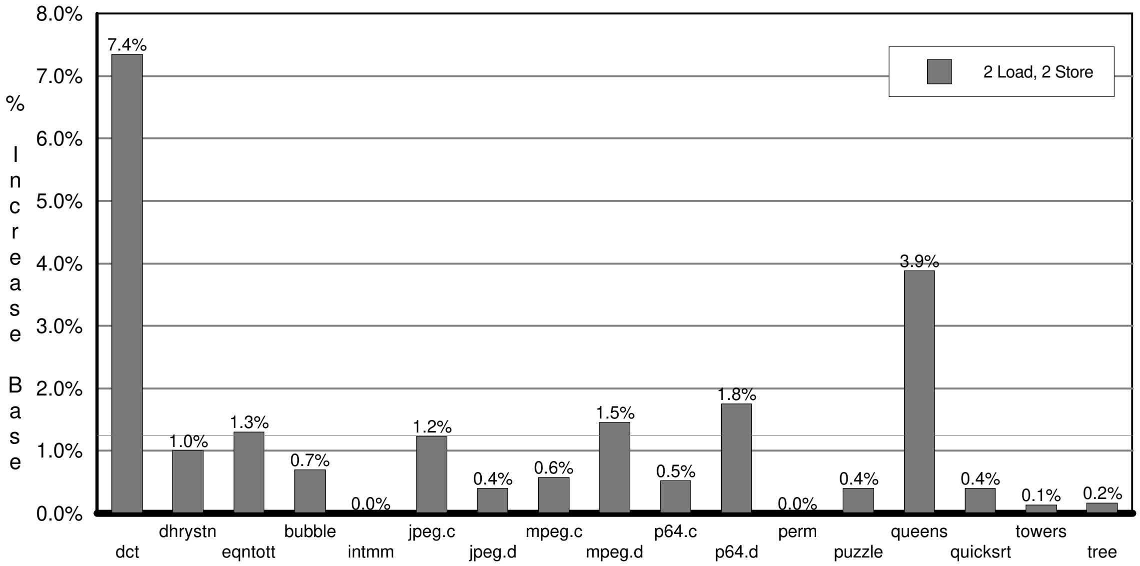

Figure 4.12: Performance Increase Using 2 Load and 2 Store Units

4.3.4 Data Cache

Data must be quickly accessible, or else poor performance will result since there will be nothing to compute. A data cache provides a one cycle load and store operation. It, however, must perform satisfactorily when data cache misses are encountered. A data cache miss on a load effects the rest of the processor since the result value will not be available for further computation by other instructions. This can cause the Scheduling Unit to eventually stall. Using a write through data cache with a 6 cycle miss penalty, Figure 4.13 shows data cache miss ratio from 1 KB to 16 KB cache size. The programs not shown had less than a 0.1% miss ratio. 1 KB to 4 KB is unacceptable; 8 KB or 16 KB yields a reasonable miss ratio. Unlike the instruction cache, the data cache has a much higher overall miss ratio for the same size cache. A 16 KB cache size is preferred since it lowers the miss ratio for most cases below 2%.

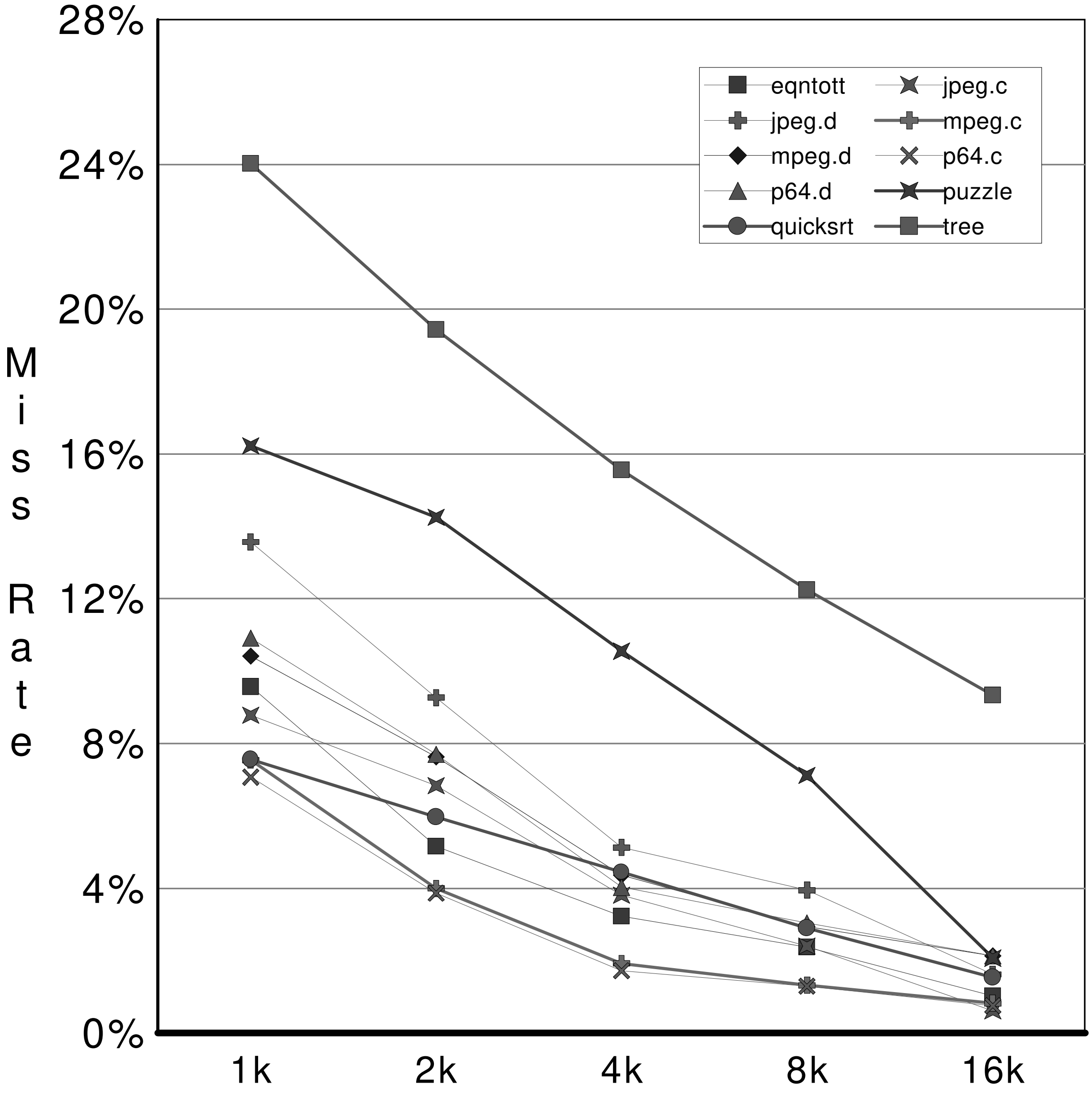

Figure 4.13: Data Cache Miss Ratio vs Size

4.4 Overall Analysis

The overall performance is determined by several factors:

- Instruction cache miss cycles, $Delay_{i-cache}$.
- Cycles delayed from data cache misses, $Delay_{d-cache}$.
- Bad branch prediction penalty cycles, $Delay_{branch}$.
- Scheduling Unit stall cycles, $Stall_{SU}$.
- Loss of potential instructions from fetch inefficiency, $Loss_{fetch}$.

The Instructions Per Cycle, IPC, is related to the size of the instruction fetch block, $Size_{block}$, and $Utilization$ by

$$IPC = Size_{block} \times Utilization$$

where

$$Utilization = 100\% - \%Delay_{i-cache} - \%Delay_{d-cache} - \%Delay_{branch} - \%Stall_{SU} - \%Loss_{fetch}$$

or equivalently related to the Average Instruction Fetch, AIF, by

$$IPC = AIF \times (100\% - \%Delay_{i-cache} - \%Delay_{d-cache} - \%Delay_{branch} - \%Stall_{SU})$$

All percentages are relative to the total number of cycles. These percentages change dynamically, but should not vary widely. The factors are mutually independent, except for some dependence between cache delays and SU stalls.

Table 4.2 shows the distribution of these factors that contribute to performance degradation. All programs used default parameters except with an 8 KB instruction cache and 8 KB data cache. Appendix E gives detailed results from the original output of the simulator for this case.

Figure 4.14 is a pie chart showing the factors contributing to overall performance. On average, the delays cause by instruction and data cache misses were less than 5%. Only 5% of the time was there a Scheduling Units stall. There was roughly a 9% penalty due to mispredicted branches. A loss in speedup due to fetching inefficiently was the greatest factor, leaving 64% utilization of a potential four times speedup.

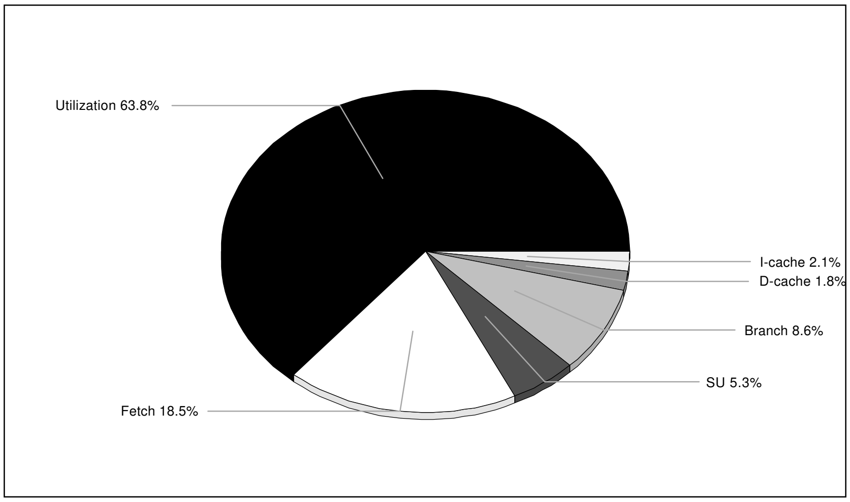

Figure 4.14: Overall Performance Factors

Program	AIF	%Icache	%Dcache	%Bra	%SU	%Fetch	IPC
dct	3.90	0.64	0.00	0.5	22.7	1.9	2.97
dhrystone	3.02	0.03	0.00	9.8	2.9	21.4	2.64
eqntott	3.16	3.55	1.08	4.7	4.0	18.3	2.74
bubble	3.00	0.06	0.06	11.6	0.0	22.1	2.65
intmm	3.39	0.11	2.34	0.5	8.9	13.4	2.99
jpeg.c	2.83	8.45	1.10	6.0	6.1	22.9	2.22
jpeg.d	3.67	9.14	3.15	2.2	13.8	5.8	2.64
mpeg.c	3.09	2.55	0.86	12.1	2.4	18.8	2.53
mpeg.d	2.89	5.25	1.92	8.2	6.8	21.6	2.25
p64.c	3.02	3.33	0.56	10.0	4.3	20.0	2.47
p64.d	3.01	1.68	2.17	7.3	8.5	19.8	2.42
perm	3.03	0.05	0.00	5.8	0.0	22.7	2.86
puzzle	2.79	0.03	2.89	8.7	3.0	25.8	2.38
queens	3.35	0.08	0.01	36.8	0.2	10.2	2.11
quicksort	2.86	0.10	2.00	11.2	1.2	24.3	2.45
towers	3.09	0.07	0.01	0.6	0.1	22.5	3.07
tree	2.76	0.05	12.5	10.4	4.5	22.5	2.52
Average	3.11	2.07	1.80	8.6	5.3	18.5	2.58

Table 4.2: Performance Factors

Chapter 5

Conclusion

This thesis presented the SDSP architecture and analyzed its performance using different configurations. The primary goal of this thesis project is to define a superscalar architecture with VLSI design consideration and investigate its performance issues. Once the performance impact from an architectural issue is known, the decision can be made if the cost in implementation is worth the performance tradeoff. Issues relating to the SDSP's Instruction Unit, Scheduling Unit, and Execution Unit were examined separately. Overall, only 64% of the maximum speedup was able to be utilized.

One critical factor in the performance of any superscalar is how control flow is handled. Using a multiple branch predictor over a simple "not taken" policy significantly increased the accuracy of instruction flow. An even more significant improvement was achieved by reducing the average bad branch penalty. This was accomplished by correcting program flow and discarding the speculative execution as soon as the branch is resolved. As a result, it is possible to have no penalty associated with a mispredicted branch and an average penalty less than one cycle. Handling branches effectively is paramount to reasonable performance, so adopting a good branch prediction method, such as presented, is pivotal.

Increasing the SU depth from 2 to 8 increases speedup considerably. Continuing to increase it to 16 provided little or no benefit for most programs, except for programs with high fetch efficiency and high ALU content. If bypassing of results is not used, a SU twice as deep is needed for equivalent performance. Furthermore, increasing the issue limit beyond the decode size increases performance. The result write limit also needs to be increased slightly, or else the full benefit from increasing the issue limit will not be realized.

Digital signal processing programs benefited most using an additional multiplier or ALU unit. All programs marginally improved using an extra load and store unit. The exact resource configuration is really dependent on the types of applications primarily used on the microprocessor; the base configuration provided a good overall performance to cost ratio.

Since the SU shifts in a FIFO manner, and most of the time a good portion of it goes unused or contains unnecessary information, it would be interesting to find out how to "compress" the SU, keeping only relevant information to maximize the ratio of parallelism extracted to hardware usage. The most important issue is the average fetch efficiency because it is the greatest factor limiting performance. The solution to this problem is static scheduling of the program so that control transfer instructions are aligned on decode boundaries. Some research into this has already been done [3], but since the solution is a NP-complete problem, new heuristics need to be developed to significantly increase AIF within a reasonable period of time.

Bibliography

[1] A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor, and N. Bagherzadeh. "VLSI Design of the TinyRISC Processor,". In *Proceedings of the 1992 IEEE Custom Integrated Circuits Conference*, pages 30.4.1–30.4.5, 1992.

[2] Mark J. Blair. *Asdsp Users's Guide*. University of California, Irvine, 1992.

[3] William Brown. "Design and Analysis of a Post-pass Scheduler for a Superscalar Digital Signal Processor,". Master's thesis, University of California, Irvine, 1993.

[4] John Ellis. *Bulldog: A compiler for VLIW Architectures*. The MIT Press, Cambridge, 1986.

[5] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadeh. "VIPER: A 25MHz, 100 MIPS Peak VLIW Microprocessor,". In *Proceedings of the 1993 IEEE Custom Integrated Circuits Conference*, San Diego, 1993.

[6] John Hennessey and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, 1990.

[7] Sheng-Chieh Huang. "Analysis and Design of a Dynamic Instruction Unit for a Superscalar Computer Architecture,". Master's thesis, University of California, Irvine, 1993.

[8] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, 1991.

[9] P. M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, New York, 1981.

[10] Johnny K. F. Lee and Alan J. Smith. "Branch Prediction Strategies and Branch Target Buffer Design,". *IEEE Computer*, pages 6–22, 1984.

[11] J. Lenell, S. Wallace, and N. Bagherzadeh. "A 20 MHz CMOS Reorder Buffer for a Superscalar Microcomputer,". In *4th Annual NASA VLSI Symposium*, November 1992.

[12] John Lenell and Nader Bagherzadeh. "A Performance Comparison of Several Superscalar Processor Models with a VLIW Processor,". In *Proceedings of International Parallel Processing Symposium*, pages 44–48, 1993.

[13] Reynold Leong. "Architecture and VHDL Modeling of a Supserscalar Microprocessor,". Master's thesis, University of California, Irvine, 1993.

[14] Motorola Inc. *MC88110 Second Generation RISC Microprocessor User's Manual*, 1991.

[15] J. E. Smith and A. R. Pleszkun. "Implementing Precise Interrupts in Pipelined Processors,". *IEEE Transactions on Computers*, C-37:562–573, May 1988.

[16] J. E. Thorton. *Design of a Computer: CDC 6600.* Scott, Foresman, Blenview, IL, 1970.

[17] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units,". *IBM Journal of research and Development*, January 1967.

[18] Shlomo Weiss and James Smith. "Instruction Issue Logic in Pipelined Supercomputers,". *IEEE Transactions on Computers*, c-33(11):1013–1022, November 1984.

Appendix A

Porting GNU CC to the SDSP

The following are instructions on how to port GNU CC 1.42 to compile SDSP code.

1. First anonymous ftp the original source code from prep.ai.mit.edu. Get the file /pub/gnu/gcc-1.42.tar.gz and extract it.

2. Ftp the ported tm.h and md configuration files from ece.uci.edu. Get the file /pub/swallace/thesis/gcc-sdsp.tar.gz and extract it.

3. Change the config.h file if the host system is not on a SUN 4 and make the compiler.

Appendix B

Porting GNU GAS to the SDSP

The following are instructions on how to port GNU GAS assembler to the SDSP.

1. First anonymous ftp the original source code from prep.ai.mit.edu. Get the file /pub/gnu/gas-1.38.1.tar.gz and extract it.

2. Ftp the sdsp.c and sdsp.h machine dependent and other modified files from ece.uci.edu in /pub/swallace/thesis/gas-sdsp.tar.gz and extract it.

3. Make the assembler.

Appendix C

Porting GNU ld to the SDSP

The following are instructions on how to port GNU "ld" to link SDSP object modules:

1. Ftp the modified source code from ece.uci.edu in /pub/swallace/thesis/ld-sdsp.tar.gz and extract it.

2. Make the linker.

Appendix D

Obtaining SDSP Simulator Source Code

The following are instructions on how to obtain SDSP simulator code.

1. Ftp the source code from ece.uci.edu in /pub/swallace/thesis/ss.tar.gz and extract it.

2. Make the simulator.

Appendix E

Detailed Results

This appendix contains detailed result information as obtained from the SDSP simulator. The following pages list the statistical output on all programs using the default parameters with an 8 KB instruction cache and 8 KB data cache.

dct

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  186579
TOTAL INTEGER CYCLES:  186579 100.00000%
TOTAL FLOATING POINT CYCLES:  0 0.00000%
TOTAL SU STALLS:  42375 22.71156%
TOTAL BRANCH DELAYS:  898 0.48130%
TOTAL ICACHE DELAYS:  1200 0.64316%
TOTAL DCACHE DELAYS:  0 0.00000%
TOTAL DELAYS:  44473 23.83602%
INSTRUCTION CACHE MISS RATIO:  200/142108 = 0.140738%.
DATA CACHE MISS RATIO:  12/54301 = 0.022099%.
TOTAL WRITE OVERFLOWS:  11983 6.42248%
ISSUE #       0      1      2      3      4      5      6   7   8
COUNT :    4071   9159  50586  69832  33856  15179   3895   0   0
    % :  2.18% 4.91% 27.11% 37.43% 18.15% 8.14% 2.09% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :    2705  17632  50070  61947  54224
    % :  1.45% 9.45% 26.84% 33.20% 29.06%
BRANCH #       0      1      2      3      4
COUNT :  137205   4902      0      0      0
    % :  2798.96% 100.00% 0.00% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0   2401   1704   3299 134703
    % :  0.00% 1.69% 1.20% 2.32% 94.79%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  447010  19200  54301  27103   6902      0
    % :  80.61% 3.46% 9.79% 4.89% 1.24% 0.00%
BRANCH PREDICTION RATE = 1 - 399/4902 = 91.86%
AVERAGE BAD BRANCH PENALTY = 898/399 = 2.3
AVERAGE RESULT WRITE:  2.7898
INSTRUCTION COUNT:  554518
AVERAGE FETCH COUNT = 554518/142106 = 3.9021
FETCH DEFICIT, LOSS:  2.44641%  1.86329%
IPC (Int & FP): 2.9720
IPC (Integer):  2.9720
SPEEDUP:  2.9720
```

dhrystone

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  3481347
TOTAL INTEGER CYCLES:  3481347 100.00000%
TOTAL FLOATING POINT CYCLES:  0 0.00000%
TOTAL SU STALLS:  99995 2.87231%
TOTAL BRANCH DELAYS:  340013 9.76671%
TOTAL ICACHE DELAYS:  1188 0.03412%
TOTAL DCACHE DELAYS:  0 0.00000%
TOTAL DELAYS:  441196 12.67314%
INSTRUCTION CACHE MISS RATIO:  198/3100158 = 0.006387%.
DATA CACHE MISS RATIO:  22/1300043 = 0.001692%.
TOTAL WRITE OVERFLOWS:  119994 3.44677%
ISSUE #       0      1      2      3      4      5   6     7     8
COUNT :    1007 740257 900101 980016 599976 259989   0     0     0
    % :  0.03% 21.26% 25.85% 28.15% 17.23% 7.47% 0.00% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :  361089 1000212 980078 859978 279989
    % :  10.37% 28.73% 28.15% 24.70% 8.04%
BRANCH #       0      1      2      3      4
COUNT :  2540154 560003      0      0      0
    % :  453.60% 100.00% 0.00% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0 620048 440042 480042 1560025
    % :  0.00% 20.40% 14.47% 15.79% 51.31%
TYPE:      ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  5280218  80000 1300043 1140021 1380074      0
    % :  57.52% 0.87% 14.16% 12.42% 15.03% 0.00%
BRANCH PREDICTION RATE = 1 - 140007/560003 = 75.00%
AVERAGE BAD BRANCH PENALTY = 340013/140007 = 2.4
AVERAGE RESULT WRITE:  1.9131
INSTRUCTION COUNT:  9180358
AVERAGE FETCH COUNT = 9180358/3040151 = 3.0197
FETCH DEFICIT, LOSS:  24.50738%  21.40153%
IPC (Int & FP): 2.6370
IPC (Integer):  2.6370
SPEEDUP:  2.6370
```

eqntott

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  1026534
TOTAL INTEGER CYCLES:  1026534 100.00000%
TOTAL FLOATING POINT CYCLES:  0 0.00000%
TOTAL SU STALLS:  40641 3.95905%
TOTAL BRANCH DELAYS:  48365 4.71149%
TOTAL ICACHE DELAYS:  36492 3.55487%
TOTAL DCACHE DELAYS:  11055 1.07692%
TOTAL DELAYS:  136553 13.30234%
INSTRUCTION CACHE MISS RATIO:  6082/891927 = 0.681894%.
DATA CACHE MISS RATIO:  7644/322526 = 2.370041%.
TOTAL WRITE OVERFLOWS:  18530 1.80510%
ISSUE #       0      1      2      3      4      5      6    7    8
COUNT :   40928  99464 232041 405189 222245  17572   9088    6    0
    % :  3.99% 9.69% 22.60% 39.47% 21.65% 1.71% 0.89% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :   56139 126924 344599 363372 135499
    % :  5.47% 12.36% 33.57% 35.40% 13.20%
BRANCH #       0      1      2      3      4
COUNT :  622666 262294   6966      0      0
    % :  225.42% 94.96% 2.52% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0  57074 207817 170325 456710
    % :  0.00% 6.41% 23.35% 19.14% 51.32%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  2125132    576 322526  53029 309260      0
    % :  75.61% 0.02% 11.48% 1.89% 11.00% 0.00%
BRANCH PREDICTION RATE = 1 - 11641/276226 = 95.79%
AVERAGE BAD BRANCH PENALTY = 48365/11641 = 4.2
AVERAGE RESULT WRITE:  2.3850
INSTRUCTION COUNT:  2810523
AVERAGE FETCH COUNT = 2810523/889981 = 3.1580
FETCH DEFICIT, LOSS:  21.05104%  18.25076%
IPC (Int & FP): 2.7379
IPC (Integer):  2.7379
SPEEDUP:  2.7379
```

bubble

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  574349
TOTAL INTEGER CYCLES:  574213 99.97632%
TOTAL FLOATING POINT CYCLES:  136 0.02368%
TOTAL SU STALLS:  259 0.04511%
TOTAL BRANCH DELAYS:  66669 11.61050%
TOTAL ICACHE DELAYS:  348 0.06060%
TOTAL DCACHE DELAYS:  373 0.06496%
TOTAL DELAYS:  67649 11.78117%
INSTRUCTION CACHE MISS RATIO:  58/506567 = 0.011450%.
DATA CACHE MISS RATIO:  135/377771 = 0.035736%.
TOTAL WRITE OVERFLOWS:  0 0.00000%
ISSUE #       0      1      2      3      4      5      6      7   8
COUNT :     581 111498 151145 138413 172572      3      0      0   0
    % :  0.10% 19.42% 26.32% 24.10% 30.05% 0.00% 0.00% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :   17684 193855 146013 216659      1
    % :  3.08% 33.76% 25.43% 37.73% 0.00%
BRANCH #       0      1      2      3      4
COUNT :  254565 252001      0      0      0
    % :  101.02% 100.00% 0.00% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0   2519 187298 124756 191993
    % :  0.00% 0.50% 36.97% 24.63% 37.90%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  757087   1004 377771 129950 253518     25
    % :  49.83% 0.07% 24.86% 8.55% 16.69% 0.00%
BRANCH PREDICTION RATE = 1 - 16712/252001 = 93.37%
AVERAGE BAD BRANCH PENALTY = 66669/16712 = 4.0
AVERAGE RESULT WRITE:  1.9781
INSTRUCTION COUNT:  1519355
AVERAGE FETCH COUNT = 1519355/506564 = 2.9993
FETCH DEFICIT, LOSS:  25.01663%  22.06938%
IPC (Int & FP): 2.6454
IPC (Integer):  2.6460
SPEEDUP:  2.6463
```

intmm

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  332648
TOTAL INTEGER CYCLES:  332512 99.95912%
TOTAL FLOATING POINT CYCLES:  136 0.04088%
TOTAL SU STALLS:  29594 8.90013%
TOTAL BRANCH DELAYS:  1732 0.52088%
TOTAL ICACHE DELAYS:  372 0.11188%
TOTAL DCACHE DELAYS:  7769 2.33646%
TOTAL DELAYS:  39467 11.86935%
INSTRUCTION CACHE MISS RATIO:  62/293046 = 0.021157%.
DATA CACHE MISS RATIO:  5412/196834 = 2.749525%.
TOTAL WRITE OVERFLOWS:  11 0.00331%
ISSUE #       0      1      2      3      4      5     6   7     8
COUNT :    1484  26782  96684  94542  75687  36038  1294   0     0
    % :  0.45% 8.05% 29.08% 28.43% 22.76% 10.84% 0.39% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :    4119  35350 112478 140250  40314
    % :  1.24% 10.63% 33.83% 42.18% 12.12%
BRANCH #       0      1      2      3      4
COUNT :  224202  68763     80      0      0
    % :  325.29% 99.77% 0.12% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0  11224  70379   3247 208195
    % :  0.00% 3.83% 24.02% 1.11% 71.05%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  509354 136124 196834  73626  78540     25
    % :  51.22% 13.69% 19.79% 7.40% 7.90% 0.00%
BRANCH PREDICTION RATE = 1 - 3318/68923 = 95.19%
AVERAGE BAD BRANCH PENALTY = 1732/3318 = 0.5
AVERAGE RESULT WRITE:  2.5332
INSTRUCTION COUNT:  994503
AVERAGE FETCH COUNT = 994503/293045 = 3.3937
FETCH DEFICIT, LOSS:  15.15783%  13.35869%
IPC (Int & FP): 2.9897
IPC (Integer):  2.9909
SPEEDUP:  2.9913
```

jpeg.c

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  4703768
TOTAL INTEGER CYCLES:  4703768 100.00000%
TOTAL FLOATING POINT CYCLES:  0 0.00000%
TOTAL SU STALLS:  289258 6.14950%
TOTAL BRANCH DELAYS:  282085 5.99700%
TOTAL ICACHE DELAYS:  397254 8.44544%
TOTAL DCACHE DELAYS:  51812 1.10150%
TOTAL DELAYS:  1020409 21.69344%
INSTRUCTION CACHE MISS RATIO:  66209/3703475 = 1.787753%.
DATA CACHE MISS RATIO:  45755/1902172 = 2.405408%.
TOTAL WRITE OVERFLOWS:  55687 1.18388%
ISSUE #       0      1       2      3      4      5     6   7     8
COUNT :  353214 740747 1969883 976277 520079 133570 9991   6     0
    % :  7.51% 15.75% 41.88% 20.76% 11.06% 2.84% 0.21% 0.00% 0.00%
WRITE #       0      1       2      3      4
COUNT :  543125 1511978 1747144 633124 268396
    % :  11.55% 32.14% 37.14% 13.46% 5.71%
BRANCH #       0      1       2      3      4
COUNT :  2390618 1268982  43874      0      0
    % :  176.20% 93.53% 3.23% 0.00% 0.00%
FETCH #       0      1       2      3      4
COUNT :       0 730874 797326 609124 1566150
    % :  0.00% 19.84% 21.65% 16.54% 42.52%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  5797925 279125 1902172 747471 1690805      0
    % :  55.66% 2.68% 18.26% 7.18% 16.23% 0.00%
BRANCH PREDICTION RATE = 1 - 114984/1356730 = 91.52%
AVERAGE BAD BRANCH PENALTY = 282085/114984 = 2.5
AVERAGE RESULT WRITE:  1.6963
INSTRUCTION COUNT:  10417498
AVERAGE FETCH COUNT = 10417498/3683359 = 2.8283
FETCH DEFICIT, LOSS:  29.29349%  22.93873%
IPC (Int & FP): 2.2147
IPC (Integer):  2.2147
SPEEDUP:  2.2147
```

jpeg.d

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  2426124
TOTAL INTEGER CYCLES:  2426118 99.99975%
TOTAL FLOATING POINT CYCLES:  6 0.00025%
TOTAL SU STALLS:  334123 13.77192%
TOTAL BRANCH DELAYS:  52658 2.17046%
TOTAL ICACHE DELAYS:  221802 9.14226%
TOTAL DCACHE DELAYS:  76308 3.14527%
TOTAL DELAYS:  684891 28.22991%
INSTRUCTION CACHE MISS RATIO:  36967/1742144 = 2.121926%.
DATA CACHE MISS RATIO:  31855/807050 = 3.947091%.
TOTAL WRITE OVERFLOWS:  114600 4.72360%
ISSUE #       0      1      2      3      4      5    6    7     8
COUNT :  208834 321820 503998 656786 580417 149030 5100  132     0
    % :  8.61% 13.26% 20.77% 27.07% 23.92% 6.14% 0.21% 0.01% 0.00%
WRITE #       0      1      2      3      4
COUNT :  216526 376893 573285 669004 590409
    % :  8.92% 15.53% 23.63% 27.58% 24.34%
BRANCH #       0      1      2      3      4
COUNT :  1569103 172187    853      0      0
    % :  902.34% 99.02% 0.49% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0 123013  70811  59393 1488926
    % :  0.00% 7.06% 4.07% 3.41% 85.51%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  5025889  59172 807050 288119 218286      2
    % :  78.55% 0.92% 12.61% 4.50% 3.41% 0.00%
BRANCH PREDICTION RATE = 1 - 13320/173893 = 92.34%
AVERAGE BAD BRANCH PENALTY = 52658/13320 = 4.0
AVERAGE RESULT WRITE:  2.4286
INSTRUCTION COUNT:  6398518
AVERAGE FETCH COUNT = 6398518/1741227 = 3.6747
FETCH DEFICIT, LOSS:  8.13205%  5.83638%
IPC (Int & FP): 2.6373
IPC (Integer):  2.6373
SPEEDUP:  2.6374
```

mpeg.c

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  775588558
TOTAL INTEGER CYCLES:  595606473 76.79413%
TOTAL FLOATING POINT CYCLES:  179982085 23.20587%
TOTAL SU STALLS:  14394052 2.41671%
TOTAL BRANCH DELAYS:  72096634 12.10474%
TOTAL ICACHE DELAYS:  15208044 2.55337%
TOTAL DCACHE DELAYS:  5128760 0.86110%
TOTAL DELAYS:  106827490 17.93592%
INSTRUCTION CACHE MISS RATIO:  2534674/491021463 = 0.516204%.
DATA CACHE MISS RATIO:  2978210/224804779 = 1.324798%.
TOTAL WRITE OVERFLOWS:  45803483 7.69023%
ISSUE #         0         1         2        3         4        5
COUNT :  28180683 144062015 117663334 170578997 84983550 48019698
                  6      7      8
             2114442   3753      0
    % :  4.73% 24.19% 19.76% 28.64% 14.27% 8.06% 0.36% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :  51748155 170118273 96342824 173935373 103461847
    % :  8.69% 28.56% 16.18% 29.20% 17.37%
BRANCH #       0      1      2      3      4
COUNT :  368621143 121628235 772084      0      0
    % :  299.27% 98.75% 0.63% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :        0 100054316 40464092 73996753 276506301
    % :  0.00% 20.47% 8.28% 15.14% 56.57%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  1062633632 11019017 224804779 22162863 163250996 25126676
    % :  70.42% 0.73% 14.90% 1.47% 10.82% 1.67%
BRANCH PREDICTION RATE = 1 - 20034070/123172403 = 83.73%
AVERAGE BAD BRANCH PENALTY = 72096634/20034070 = 3.6
AVERAGE RESULT WRITE:  2.1801
INSTRUCTION COUNT:  1508997963
AVERAGE FETCH COUNT = 1508997963/488778983 = 3.0873
FETCH DEFICIT, LOSS:  22.81798%  -23.32592%
IPC (Int & FP): 1.9456
IPC (Integer):  2.5335
SPEEDUP:  -2.6638
```

mpeg.d

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  57594460
TOTAL INTEGER CYCLES:  57594460 100.00000%
TOTAL FLOATING POINT CYCLES:  0 0.00000%
TOTAL SU STALLS:  3896620 6.76562%
TOTAL BRANCH DELAYS:  4733393 8.21849%
TOTAL ICACHE DELAYS:  3023370 5.24941%
TOTAL DCACHE DELAYS:  1104606 1.91790%
TOTAL DELAYS:  12757989 22.15142%
INSTRUCTION CACHE MISS RATIO:  503895/44948177 = 1.121058%.
DATA CACHE MISS RATIO:  633536/21461211 = 2.952005%.
TOTAL WRITE OVERFLOWS:  1389616 2.41276%
ISSUE #       0      1      2      3      4      5      6
COUNT :  4606400 11953267 17227408 16036999 4819965 1989745 941443
    % :  8.00% 20.75% 29.91% 27.84% 8.37% 3.45% 1.63%
      7      8
  19232      0
WRITE #       0      1      2      3      4
COUNT :  6187744 20013519 18330374 8380293 4682529
    % :  10.74% 34.75% 31.83% 14.55% 8.13%
BRANCH #       0      1      2      3      4
COUNT :  27934486 16914809  98881      0      0
    % :  163.24% 98.84% 0.58% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :        0 8831727 6623760 10519056 18973633
    % :  0.00% 19.70% 14.77% 23.46% 42.32%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  76459654 2624397 21461211 8624289 20361396      0
    % :  59.03% 2.03% 16.57% 6.66% 15.72% 0.00%
BRANCH PREDICTION RATE = 1 - 1398294/17112571 = 91.83%
AVERAGE BAD BRANCH PENALTY = 4733393/1398294 = 3.4
AVERAGE RESULT WRITE:  1.7457
INSTRUCTION COUNT:  129530947
AVERAGE FETCH COUNT = 129530947/44836471 = 2.8890
FETCH DEFICIT, LOSS:  27.77590%  21.62315%
IPC (Int & FP): 2.2490
IPC (Integer):  2.2490
SPEEDUP:  2.2490
```

p64.c

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  353579273
TOTAL INTEGER CYCLES:  271503264 76.78710%
TOTAL FLOATING POINT CYCLES:  82076009 23.21290%
TOTAL SU STALLS:  11627368 4.28259%
TOTAL BRANCH DELAYS:  27152177 10.00068%
TOTAL ICACHE DELAYS:  9047226 3.33227%
TOTAL DCACHE DELAYS:  1519417 0.55963%
TOTAL DELAYS:  49346188 18.17517%
INSTRUCTION CACHE MISS RATIO:  1507871/223113358 = 0.675832%.
DATA CACHE MISS RATIO:  1335497/103572475 = 1.289432%.
TOTAL WRITE OVERFLOWS:  17372815 6.39875%
ISSUE #        0         1         2         3         4         5      6
COUNT :  13697665 49675688 75513908 90887677 21935807 18804085 982545
    % :      5.05%    18.30%    27.81%    33.48%     8.08%     6.93%  0.36%
 7      8
      5888      0
WRITE #        0      1      2      3      4
COUNT :  22384666 72336512 66095932 80385628 30300525
    % :  8.24% 26.64% 24.34% 29.61% 11.16%
BRANCH #        0      1      2      3      4
COUNT :  162682880 59961469 469008       0       0
    % :  267.13% 98.46% 0.77% 0.00% 0.00%
FETCH #        0      1      2      3      4
COUNT :        0 38014644 30783787 45070962 109243964
    % :  0.00% 17.11% 13.86% 20.29% 49.17%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  454055864 9259021 103572475 16619087 77560227 10704286
    % :  67.59% 1.38% 15.42% 2.47% 11.55% 1.59%
BRANCH PREDICTION RATE = 1 - 8086392/60899485 = 86.72%
AVERAGE BAD BRANCH PENALTY = 27152177/8086392 = 3.4
AVERAGE RESULT WRITE:  2.0880
INSTRUCTION COUNT:  671770960
AVERAGE FETCH COUNT = 671770960/222157076 = 3.0239
FETCH DEFICIT, LOSS:  24.40361%  19.96821%
IPC (Int & FP): 1.8999
IPC (Integer):  2.4743
SPEEDUP:  2.8284
```

p64.d

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  41379986
TOTAL INTEGER CYCLES:  41379986 100.00000%
TOTAL FLOATING POINT CYCLES:  0 0.00000%
TOTAL SU STALLS:  3526828 8.52303%
TOTAL BRANCH DELAYS:  3040838 7.34857%
TOTAL ICACHE DELAYS:  694380 1.67806%
TOTAL DCACHE DELAYS:  899212 2.17306%
TOTAL DELAYS:  8161258 19.72272%
INSTRUCTION CACHE MISS RATIO:  115730/33285781 = 0.347686%.
DATA CACHE MISS RATIO:  505988/16883599 = 2.996920%.
TOTAL WRITE OVERFLOWS:  1416071 3.42212%
ISSUE #       0        1        2        3       4       5      6
COUNT :  2016976 7306492 12957693 13031875 3493060 1651994 919061
    % :    4.87%  17.66%   31.31%   31.49%   8.44%   3.99%  2.22%
      7      8
   2834      0
WRITE #       0      1      2      3      4
COUNT :  3065093 13029178 14443909 6708367 4133438
    % :  7.41% 31.49% 34.91% 16.21% 9.99%
BRANCH #       0      1      2      3      4
COUNT :  21129988 12131947  23845      0      0
    % :  173.49% 99.61% 0.20% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :        0 5359529 4728678 7523260 15674313
    % :  0.00% 16.13% 14.23% 22.65% 47.19%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  59448522 2243728 16883599 6993268 14514800      0
    % :  59.40% 2.24% 16.87% 6.99% 14.50% 0.00%
BRANCH PREDICTION RATE = 1 - 897039/12179637 = 92.63%
AVERAGE BAD BRANCH PENALTY = 3040838/897039 = 3.4
AVERAGE RESULT WRITE:  1.8989
INSTRUCTION COUNT:  100083917
AVERAGE FETCH COUNT = 100083917/33218728 = 3.0129
FETCH DEFICIT, LOSS:  24.67809%  19.81090%
IPC (Int & FP): 2.4187
IPC (Integer):  2.4187
SPEEDUP:  2.4187
```

perm

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  693739
TOTAL INTEGER CYCLES:  693603 99.98040%
TOTAL FLOATING POINT CYCLES:  136 0.01960%
TOTAL SU STALLS:  7 0.00101%
TOTAL BRANCH DELAYS:  40422 5.82783%
TOTAL ICACHE DELAYS:  336 0.04844%
TOTAL DCACHE DELAYS:  1 0.00014%
TOTAL DELAYS:  40766 5.87743%
INSTRUCTION CACHE MISS RATIO:  56/652842 = 0.008578%.
DATA CACHE MISS RATIO:  16/360603 = 0.004437%.
TOTAL WRITE OVERFLOWS:  0 0.00000%
ISSUE #       0      1      2      3      4      5   6     7     8
COUNT :     300  20811 193279 349310 124402   5500  0     0     0
    % :  0.04% 3.00% 27.87% 50.36% 17.94% 0.79% 0.00% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :   14226 193049 315995 160629   9703
    % :  2.05% 27.83% 45.56% 23.16% 1.40%
BRANCH #       0      1      2      3      4
COUNT :  566202  86639      0      0      0
    % :  653.52% 100.00% 0.00% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0 118951  86633 100813 346444
    % :  0.00% 18.22% 13.27% 15.44% 53.07%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  985131      4 360603 360629 274040     25
    % :  49.74% 0.00% 18.21% 18.21% 13.84% 0.00%
BRANCH PREDICTION RATE = 1 - 22313/86639 = 74.25%
AVERAGE BAD BRANCH PENALTY = 40422/22313 = 1.8
AVERAGE RESULT WRITE:  1.9402
INSTRUCTION COUNT:  1980432
AVERAGE FETCH COUNT = 1980432/652837 = 3.0336
FETCH DEFICIT, LOSS:  24.16055%  22.74053%
IPC (Int & FP): 2.8547
IPC (Integer):  2.8553
SPEEDUP:  2.8555
```

puzzle

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  3551575
TOTAL INTEGER CYCLES:  3551439 99.99617%
TOTAL FLOATING POINT CYCLES:  136 0.00383%
TOTAL SU STALLS:  106206 2.99051%
TOTAL BRANCH DELAYS:  309363 8.71092%
TOTAL ICACHE DELAYS:  1134 0.03193%
TOTAL DCACHE DELAYS:  102511 2.88646%
TOTAL DELAYS:  519214 14.61982%
INSTRUCTION CACHE MISS RATIO:  189/3032259 = 0.006233%.
DATA CACHE MISS RATIO:  118965/1671091 = 7.119002%.
TOTAL WRITE OVERFLOWS:  15437 0.43467%
ISSUE #       0      1       2       3      4      5   6     7     8
COUNT :  171492 547944 1111028 1226509 459695  34768 2     0     0
    % :  4.83% 15.43% 31.28% 34.54% 12.94% 0.98% 0.00% 0.00% 0.00%
WRITE #       0      1       2       3      4
COUNT :  220269 892505 1631636 727552  79476
    % :  6.20% 25.13% 45.94% 20.49% 2.24%
BRANCH #       0      1       2       3      4
COUNT :  1329414 1702802     42       0      0
    % :  78.07% 100.00% 0.00% 0.00% 0.00%
FETCH #       0      1       2       3      4
COUNT :       0 954521 228422 346441 1502874
    % :  0.00% 31.48% 7.53% 11.43% 49.56%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  4985242      4 1671091  44897 1760925     25
    % :  58.91% 0.00% 19.75% 0.53% 20.81% 0.00%
BRANCH PREDICTION RATE = 1 - 71508/1702886 = 95.80%
AVERAGE BAD BRANCH PENALTY = 309363/71508 = 4.3
AVERAGE RESULT WRITE:  1.8743
INSTRUCTION COUNT:  8462184
AVERAGE FETCH COUNT = 8462184/3032225 = 2.7908
FETCH DEFICIT, LOSS:  30.23123%  25.81148%
IPC (Int & FP): 2.3827
IPC (Integer):  2.3827
SPEEDUP:  2.3828
```

queens

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  598031
TOTAL INTEGER CYCLES:  597895 99.97726%
TOTAL FLOATING POINT CYCLES:  136 0.02274%
TOTAL SU STALLS:  1199 0.20054%
TOTAL BRANCH DELAYS:  219737 36.75177%
TOTAL ICACHE DELAYS:  480 0.08028%
TOTAL DCACHE DELAYS:  65 0.01087%
TOTAL DELAYS:  221481 37.04346%
INSTRUCTION CACHE MISS RATIO:  80/376423 = 0.021253%.
DATA CACHE MISS RATIO:  41/218772 = 0.018741%.
TOTAL WRITE OVERFLOWS:  0 0.00000%
ISSUE #       0      1      2      3      4      5    6    7     8
COUNT :     496 249209 125920 163369  29819  23486 5595   0     0
    % :  0.08% 41.68% 21.06% 27.32% 4.99% 3.93% 0.94% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :   67057 296315 134786  71426  28310
    % :  11.22% 49.56% 22.54% 11.95% 4.73%
BRANCH #       0      1      2      3      4
COUNT :  189169 186453    800      0      0
    % :  100.59% 99.15% 0.43% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0  26334  44355  76519 229214
    % :  0.00% 7.00% 11.78% 20.33% 60.89%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  674629      4 218772 163313 204714     25
    % :  53.48% 0.00% 17.34% 12.95% 16.23% 0.00%
BRANCH PREDICTION RATE = 1 - 66299/188053 = 64.74%
AVERAGE BAD BRANCH PENALTY = 219737/66299 = 3.3
AVERAGE RESULT WRITE:  1.4943
INSTRUCTION COUNT:  1261457
AVERAGE FETCH COUNT = 1261457/376414 = 3.3512
FETCH DEFICIT, LOSS:  16.21878%  10.21078%
IPC (Int & FP): 2.1094
IPC (Integer):  2.1098
SPEEDUP:  2.1103
```

quicksort

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  409152
TOTAL INTEGER CYCLES:  409016 99.96676%
TOTAL FLOATING POINT CYCLES:  136 0.03324%
TOTAL SU STALLS:  4921 1.20313%
TOTAL BRANCH DELAYS:  45786 11.19418%
TOTAL ICACHE DELAYS:  414 0.10122%
TOTAL DCACHE DELAYS:  8190 2.00237%
TOTAL DELAYS:  59311 14.50090%
INSTRUCTION CACHE MISS RATIO:  69/349713 = 0.019730%.
DATA CACHE MISS RATIO:  4564/157083 = 2.905470%.
TOTAL WRITE OVERFLOWS:  13 0.00318%
ISSUE #       0      1      2      3      4      5   6     7     8
COUNT :    5079  80682 143036 102716  61940  14620 942     0     0
    % :  1.24% 19.73% 34.97% 25.11% 15.14% 3.57% 0.23% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :   24930 121053 182792  51713  28527
    % :  6.10% 29.60% 44.69% 12.64% 6.97%
BRANCH #       0      1      2      3      4
COUNT :  220694 111935  17083      0      0
    % :  151.06% 76.61% 11.69% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0  78255  54236  54172 163049
    % :  0.00% 22.38% 15.51% 15.49% 46.62%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  588797  10004 157083  59024 186506     25
    % :  58.80% 1.00% 15.69% 5.89% 18.62% 0.00%
BRANCH PREDICTION RATE = 1 - 15600/146101 = 89.32%
AVERAGE BAD BRANCH PENALTY = 45786/15600 = 2.9
AVERAGE RESULT WRITE:  1.8481
INSTRUCTION COUNT:  1001439
AVERAGE FETCH COUNT = 1001439/349705 = 2.8637
FETCH DEFICIT, LOSS:  28.40830%  24.28884%
IPC (Int & FP): 2.4476
IPC (Integer):  2.4484
SPEEDUP:  2.4489
```

towers

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  793254
TOTAL INTEGER CYCLES:  793118 99.98286%
TOTAL FLOATING POINT CYCLES:  136 0.01714%
TOTAL SU STALLS:  1036 0.13062%
TOTAL BRANCH DELAYS:  4877 0.61491%
TOTAL ICACHE DELAYS:  534 0.06733%
TOTAL DCACHE DELAYS:  55 0.00693%
TOTAL DELAYS:  6502 0.81980%
INSTRUCTION CACHE MISS RATIO:  89/786620 = 0.011314%.
DATA CACHE MISS RATIO:  32/409499 = 0.007814%.
TOTAL WRITE OVERFLOWS:  16395 2.06716%
ISSUE #       0      1      2      3      4      5     6  7     8
COUNT :     499  57539 172914 296620 204857  44293 16395  0     0
    % :  0.06% 7.25% 21.80% 37.40% 25.83% 5.58% 2.07% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :    4879 126682 394211 215611  51734
    % :  0.62% 15.97% 49.70% 27.19% 6.52%
BRANCH #       0      1      2      3      4
COUNT :  688483  98136      0      0      0
    % :  701.56% 100.00% 0.00% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0 147817  73777 122956 442069
    % :  0.00% 18.79% 9.38% 15.63% 56.20%
TYPE:       ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  1359370      4 409499 360604 303013     25
    % :  55.88% 0.00% 16.83% 14.82% 12.46% 0.00%
BRANCH PREDICTION RATE = 1 - 4355/98136 = 95.56%
AVERAGE BAD BRANCH PENALTY = 4877/4355 = 1.1
AVERAGE RESULT WRITE:  2.2303
INSTRUCTION COUNT:  2432515
AVERAGE FETCH COUNT = 2432515/786616 = 3.0924
FETCH DEFICIT, LOSS:  22.69052%  22.50450%
IPC (Int & FP): 3.0665
IPC (Integer):  3.0670
SPEEDUP:  3.0672
```

tree

```
parms = 4 8 4 8 15 8176 8176 63 4 1 1 1
TOTAL CYCLES:  1024079
TOTAL INTEGER CYCLES:  1023943 99.98672%
TOTAL FLOATING POINT CYCLES:  136 0.01328%
TOTAL SU STALLS:  46338 4.52545%
TOTAL BRANCH DELAYS:  106936 10.44355%
TOTAL ICACHE DELAYS:  558 0.05450%
TOTAL DCACHE DELAYS:  127765 12.47775%
TOTAL DELAYS:  281597 27.50124%
INSTRUCTION CACHE MISS RATIO:  93/742353 = 0.012528%.
DATA CACHE MISS RATIO:  52715/430459 = 12.246230%.
TOTAL WRITE OVERFLOWS:  4454 0.43499%
ISSUE #       0      1      2      3      4      5    6     7     8
COUNT :  160106 225279 268886 217726 127905  23590  450     0     0
    % :  15.64% 22.00% 26.26% 21.26% 12.49% 2.30% 0.04% 0.00% 0.00%
WRITE #       0      1      2      3      4
COUNT :  188749 436652 215811 159706  23024
    % :  18.43% 42.64% 21.08% 15.60% 2.25%
BRANCH #       0      1      2      3      4
COUNT :  515766 224053   2533      0      0
    % :  225.11% 97.79% 1.11% 0.00% 0.00%
FETCH #       0      1      2      3      4
COUNT :       0 173740 140532 118600 309480
    % :  0.00% 23.40% 18.93% 15.98% 41.69%
TYPE:      ALU    MULT   LOAD   STOR   CNTL   FLPT
COUNT :  999025  10004 430459 145689 463322     25
    % :  48.77% 0.49% 21.01% 7.11% 22.62% 0.00%
BRANCH PREDICTION RATE = 1 - 23166/229119 = 89.89%
AVERAGE BAD BRANCH PENALTY = 106936/23166 = 4.6
AVERAGE RESULT WRITE:  1.4058
INSTRUCTION COUNT:  2048524
AVERAGE FETCH COUNT = 2048524/742346 = 2.7595
FETCH DEFICIT, LOSS:  31.01182%  22.48319%
IPC (Int & FP): 2.0004
IPC (Integer):  2.0006
SPEEDUP:  2.0009
```