

PERFORMANCE ISSUES OF A SUPERSCALAR MICROPROCESSOR

Steven Wallace and Nader Bagherzadeh
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92717
swallace or nader@ece.uci.edu

Abstract: *Cache, dynamic scheduling, bypassing, branch prediction, and fetch efficiency are primary issues concerning performance of a superscalar microprocessor. This paper considers all these issues and shows their impact on performance by running our simulator on seventeen different programs. Our approach in handling branch prediction is shown to significantly decrease the bad branch penalty. Furthermore, results show that the average instruction fetch places an upper bound on speedup and is the most critical factor in determining overall performance. Its performance impact is greater than all other factors combined.*

INTRODUCTION

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 [4]. Another approach, commonly known as a Very Long Instruction Word (VLIW) architecture [2], is entirely based on compile-time analysis to extract parallelism.

These architectures exploit ILP by issuing more than one instruction per cycle. VLIW processors require a sophisticated compiler while superscalars utilize dynamic scheduling to extract parallelism at run-time, in addition to static scheduling [6].

This paper 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 GNU CC compiler, assembler, and linker were ported. A simulator was written 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, each issue is evaluated to determine its performance impact. Finally, all factors are considered together.

THE SDSP 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. 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, 2]. Design decisions were made with full consideration for VLSI implementation. For a detailed definition of the SDSP and its instruction set, refer to [8].

Architectural Organization

The architecture organization of the SDSP is shown in Figure 1. It is divided into three basic units: the Instruction Unit (IU), the Scheduling Unit (SU), and the Execution Unit (EU).

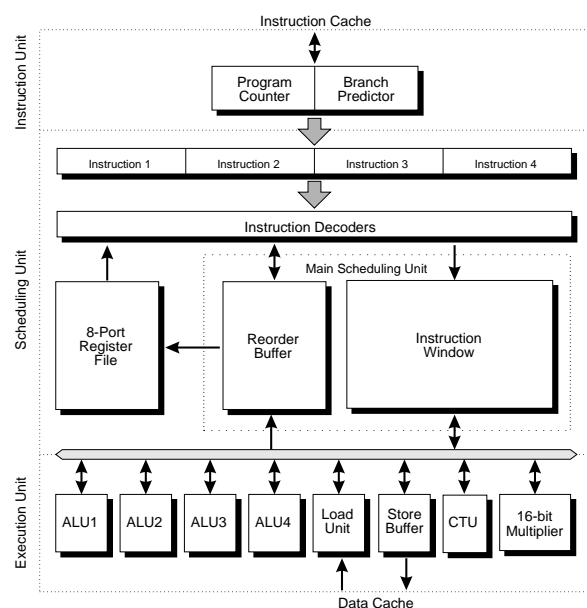


Figure 1: SDSP Architectural Organization

Instruction Unit

The IU maintains the control flow of the processor. The basic components of the IU are the Program Counter (PC) and branch predictor.

Program Counter. The PC handles the control flow of the processor. The memory address used by the PC is organized in the format shown in Figure 2.

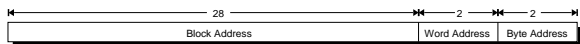


Figure 2: Memory Address Configuration

Branch Predictor. SDSP uses a superscalar branch predictor to support speculative execution [3]. The predictor is a modified branch target buffer [5], 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. 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 SU commits a block of results to the register file, branch predictor statistics are updated based on outcome of branch predictions in that block.

Scheduling Unit

The Scheduling Unit decodes and issues instructions received from the Instruction Unit to the Execution Unit. It maintains the proper state of the machine. These tasks are accomplished using an instruction window, reorder buffer, and register file [4].

Dynamic out-of-order scheduling is achieved using a reorder buffer and a central instruction window, which operate 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.

New instructions enter at the top of the SU. Completed instructions exit at the bottom of the SU and are committed to the register file. 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.

Scheduling Algorithm. The SDSP issues instructions using the “oldest first” scheduling algorithm. 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.

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.

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.

Execution Unit

The execution unit consists of four integer Arithmetic and Logic Units (ALU), one 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.

SIMULATION METHODS

Simulator

After compiling, assembling, and linking the original C source code, the object code is loaded and executed. A cycle by cycle simulation gathers statistics about its execution.

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 and no-ops.

Branch Prediction. When a branch is encountered, the simulator continues down its predicted path until it is determined to be incorrect.

Scheduling Unit. Instructions are issued out-of-order using the “oldest first” scheduling algorithm, as long as the issue limit parameter has not been 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. The penalty for a cache miss is 6 cycles.

Floating Point. FP 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. OS 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.

Benchmarks

The results in this paper are from seventeen integer-intensive programs. All benchmarks were run until completion. Eight of the programs come from the Stanford suite of benchmarks (*bubble*, *intmm*, *perm*, *puzzle*, *queens*, *quicksort*, *towers*, *tree*). *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. Execution from these programs ranges from just a half a million to one and a half *billion* instructions.

Default Configuration

Unless otherwise noted, the base configuration of the simulator uses 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.

PERFORMANCE FACTORS

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 is 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 3 shows the AIF count for each program for a decode size of 2 and a decode size of 4.

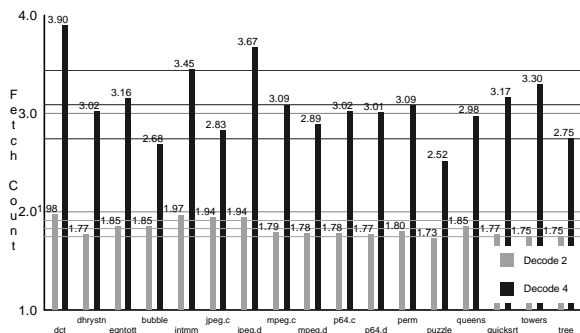


Figure 3: Average Instruction Fetch vs. Decode Size

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 fetch for a decode size of 2 and 4. Although the decode size doubled, the AIF only increased by about 73%. The average instruction loss per block due to misalignment increases as the decode size increases. Therefore, increasing the decode size has a marginal re-

turn to AIF.

For a decode size of 4, Figure 4 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.

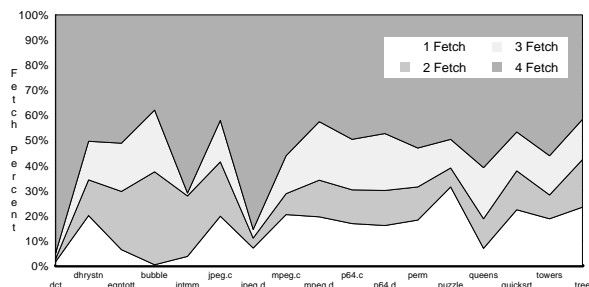


Figure 4: Instruction Fetch Distribution

Programs with less alignment usually reflect more control transfer instructions and lower AIF. In the case of DCT, the AIF is nearly perfect because instruction runs are long. If instructions could be rearranged such that most control transfer instructions are at the last decode position, and its destination address is at the first decode position, a substantial increase in AIF could be achieved.

Branch Prediction

Using a 64-entry prediction branch buffer, the programs were able to achieve an average branch prediction accuracy of 88%. 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 5.

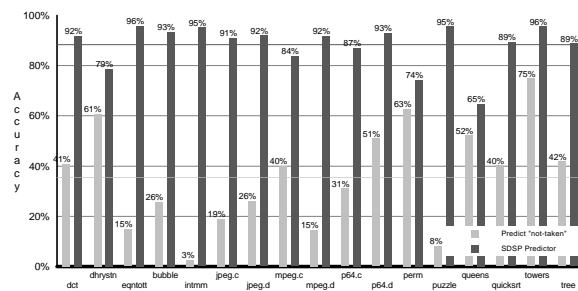


Figure 5: Branch Prediction Accuracy

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 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.

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.

Table 1: Average Bad Branch Penalty vs. SU Depth

| 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 |

SU Depth

The depth of the SU determines the potential of extracting parallelism and hiding the latency of longer latency operations (such as a data cache miss on a load operation). Figure 6 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 of 4 to 8 is significant in most cases. Comparing the speedup using SU depth of 8 to the average fetch count, as in Figure 3, 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, branch accuracy is sometimes lost when increasing the SU depth because this increases the latency required for update.

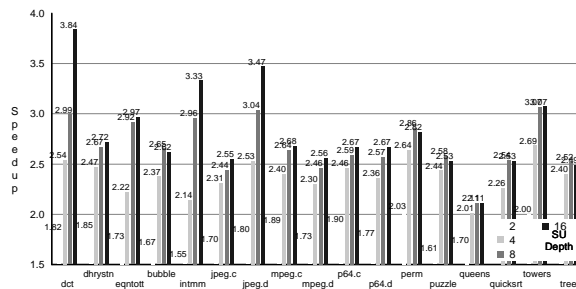


Figure 6: Speedup Improvement

Bypassing

The SDSP uses complete bypassing of results to functional units to avoid extra latency. This, however, is costly in hardware. Figure 7 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.

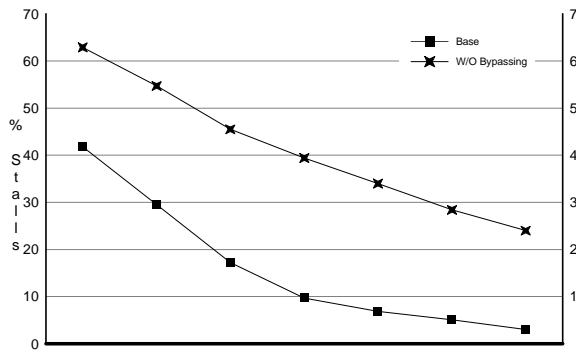


Figure 7: Percentage Stalls vs. SU Depth

Cache

Instruction fetching must be sustained, or it will be impossible to achieve a high IPC rate. In addition, data must be quickly accessible, or else poor performance will result since there will be nothing to compute. Therefore, the instruction and data cache should be designed to allow the processor to achieve close to ideal performance, ignoring other factors. With a 4 KB or 8 KB instruction and data cache, most programs ran with a cache miss rate under 1% with a total degradation in performance of about 5%, on the average. Although cache can not be ignored, its performance impact is well known and can be designed to meet the performance needs of most programs [7].

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, $Stall_{SU}$; and 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

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

All percentages are relative to the total number of cycles. These percentages change dynamically, but should not vary widely. Table 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.

Table 2: Performance Factors

| Program | AIF | %I-c | %D-c | %Br | %SU | %Fe | IPC |
|---------|------|------|------|------|------|------|------|
| dct | 3.90 | 0.64 | 0.00 | 0.5 | 22.7 | 1.9 | 2.97 |
| dhrystn | 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 |
| quickst | 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 |

Figure 8 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. Roughly, there was 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.

CONCLUSION

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 by correcting program

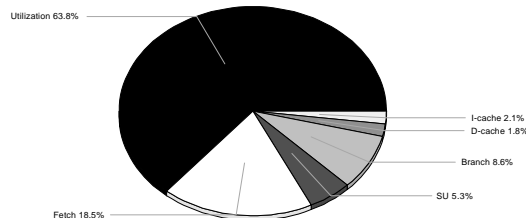


Figure 8: Overall Performance Factors

flow and discarding the speculative execution as soon as the branch is resolved. The most important issue of a superscalar microprocessor is the Average Instruction Fetch efficiency because it is the greatest factor limiting performance.

REFERENCES

- [1] A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor, and N. Bagherzadeh, "VLSI Design of the TinyRISC Processor," *Proceedings of the 1992 IEEE Custom Integrated Circuits Conference*, pp. 30.4.1–30.4.5, 1992.
- [2] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadeh, "VIPER: A 25MHz, 100 MIPS Peak VLIW Microprocessor," *Proceedings of the 1993 IEEE Custom Integrated Circuits Conference*, San Diego, 1993.
- [3] Sheng-Chieh Huang, *Analysis and Design of a Dynamic Instruction Unit for a Superscalar Computer Architecture*, Master's thesis, University of California, Irvine, 1993.
- [4] Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, 1991.
- [5] Johnny K. F. Lee and Alan J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, pp. 6–22, 1984.
- [6] John Lenell and Nader Bagherzadeh, "A Performance Comparison of Several Superscalar Processor Models with a VLIW Processor," *Proceedings of International Parallel Processing Symposium*, pp. 44–48, 1993.
- [7] Steven Przybylski, Mark Horowitz, and John Hennessy, "Performance Tradeoffs in Cache Design," *Proceedings of the 15th Annual Symposium on Computer Architecture*, pp. 290–298, 1988.
- [8] Steven Wallace, *Performance Analysis of a Superscalar Architecture*, Master's thesis, University of California, Irvine, 1993.