QUANTIFYING THE IMPACTS OF DISABLING SPECULATION AND RELAXING THE SCHEDULING LOOP IN MULTITHREADED PROCESSORS

BY

JASON LOEW

BS, Computer Science, Binghamton University, 2004
BA, Psychology, Binghamton University, 2004

THESIS

Submitted in partial fulfillment of the requirements for
the degree of Master of Science in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2006

© Copyright by Jason Loew 2006

All Rights Reserved

Accepted in partial fulfillment of the requirements for
the degree of Master of Science in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2006

Dmitry Ponomarev ____ August 2nd, 2006
Department of Computer Science, Binghamton University

Kanad Ghose____ August 2nd, 2006
Department of Computer Science, Binghamton University

Abstract

Traditional architectural approaches for increasing microprocessor performance rely on the use of large, complex, highly-speculative out-of-order cores to extract Instruction-Level Parallelism (ILP) from single-threaded applications. In order to realize high performance, these designs employ a myriad of speculative techniques, ranging from branch prediction to load-latency prediction and memory-dependence prediction. While these techniques are absolutely essential for realizing high performance in single-threaded machines, it is conceivable that they could be less important in processors that exploit Thread-Level Parallelism.

The goal of this thesis is to investigate the impact of disabling or limiting the branch and load-hit speculation, as well as the impact of pipelining the scheduling logic on the performance of SMT processors. We begin by examining the synergy of speculative execution with multithreading. If sufficient TLP exists, then disabling the speculative execution on SMT can, at least in theory, result in the allocation of resources to only non-speculative instructions thus possibly even increasing the performance. We quantify the impact of completely disallowing speculative execution as well as only resorting to speculative execution when no non-speculative instruction from any thread is available for fetching.

We then study and quantify the impact of disabling load-hit speculation on SMT machines. The motivation is that the bubbles created as a result of such restriction are likely to be filled by the load-independent instructions from other threads on SMT, thus possibly mitigating the performance impact.

Lastly, the impact of pipelining the instruction scheduling logic into separate wakeup and selection stages is quantified. Just as with load-hit speculation, if the pipeline bubbles are filled with the instructions from other threads, then the impact on the performance can be negligible. Various instruction selection schemes specifically targeted towards this goal are proposed and analyzed.

Acknowledgments

I would like to thank my advisor, Dmitry Ponomarev. His support and encouragement has been a tremendous influence towards the completion of this work and has allowed me to continue my studies. I appreciate his efforts in helping me to complete this work. I also want to thank Joseph Sharkey for being a constant source of insight; both with the simulator I worked with and his own experience within the field.

I am thankful for the experience that my masters work has given me; the opportunity to take advanced courses and the professors who guided me along the way. All of them have made my experience one that will benefit me for the rest of my life.

Finally, I want to thank my wife Maura, for being supportive when my work presented difficulties and helping me to keep my mind focused when it needed to be.

Table of Contents

1. INTRODUCTION.. 1

1.1 Goals and Contributions...8
1.2 Thesis Organization... 9

2. DATAPATH ORGANIZATION OF SIMULTANEOUSLY MULTITHREADED PROCESSORS.. 11

2.1 Organization of a Basic SMT Datapath... 11
2.2 Scheduler Design...15
2.3 Instruction Fetcher Design... 16
2.4 Branch Prediction...17
2.4.1 Branch Predictors...18

3. RELATED WORK..20

3.1 Execution Schedulers...20
3.1.1 Pipelined Scheduling with Speculative Wakeup.................................. 20
3.1.2 Select-Free Instruction Scheduling Logic... 22
3.2 Load Latency Prediction...23
3.2.1 Managing Load-Related Scheduling Issues on SMT............................24
3.2.2 Load-Hit Speculation Techniques.. 26
3.2.3 Additional Resources to Combat Load Latency Misprediction Costs.. 27
3.3 Branch Prediction...28
3.3.1 Current Trends: Pipelined Branch Predictors.................................... 29
3.4 Fetch Policies.. 30

4. SIMULATION METHODOLOGY...33

5. EVALUATING THE IMPACT OF DISABLING SPECULATIVE EXECUTION ON SMT 38

5.1 DIsabling SPeculative EXecution (DISPEX): The Implementation..............40
5.2 DISPEX with Speculative Fetch (DISPEX+SF)...41
5.3 DISPEX+SF with MICOUNT and ADEBRL (Aggressive DISPEX+SF).....43
5.4 Results of DISPEX, DISPEX+SF, ADISPEX+SF..44
5.5 DYnamically COntrolled SPeculative EXecution (DYCOSPEX).................53
5.6 Results of DYCOSPEX...54
5.7 COnfidence COntrolled SPeculative EXecution (COCOSPEX)...................59
5.8 Results of COCOSPEX..60
5.9 Summary.. 65

6. LOAD-HIT SPECULATION.. 67

6.1 Disabling Speculative Scheduling: The Implementation............................... 70
6.2 Results of Disabling Speculative Scheduling on SMT................................. 70
6.3 Summary of Disabling Speculative Scheduling on SMT..............................76

7. RELAXING SCHEDULING LOOPS ON SMT PIPELINED SELECTION LOGIC.. 78

7.1 Pipelining the Scheduling Logic Over 2 Cycles: The Implementation..........80
7.2 Results of Pipelining the Scheduling Logic Over 2 Cycles........................... 81
7.3 Summary of Pipelining Scheduling Loops Over 2 Cycles.............................86
7.4 Selection Methods...86
7.5 Implementation of Various Selection Methods...86
7.6 Results of Various Selection Methods... 88
7.7 Summary of Various Selection Methods... 94

8. CONCLUSIONS AND FUTURE WORK...97

REFERENCES... 102

List of Figures

FIGURE 1.1: Pipeline Diagram.... 2
FIGURE 1.2: Execution Schedule Without Load-Hit Speculation.... 2
FIGURE 1.3: Code Example to Examine Back-to-Back Execution.... 4
FIGURE 1.4: Pipeline Diagram for Figure 1.3 with Atomic Scheduling.... 5
FIGURE 1.5: Pipeline Diagram for Figure 1.3 with Pipelined Scheduling.... 5
FIGURE 1.6: Processor-Memory Gap.... 7
FIGURE 2.1: SMT Pipeline Diagram.... 11
FIGURE 2.2: Code Example to Examine Write-after-Write Data Hazard.... 12
FIGURE 2.3: Code Example to Examine Write-after-Read Data Hazard.... 13
FIGURE 2.4: Application of Register Rename to Data Hazards.... 13
FIGURE 3.1: Example Data Flow Graph [SBP 00].... 21
FIGURE 3.2: Execution Core of a Processor with Select-Free Scheduling [BSP 01] 23
FIGURE 3.3: Placement of the Recovery Buffer [MLO 01].... 28
FIGURE 3.4: Separation of the gshare Predictor [J 03].... 30
FIGURE 4.1: Cycle-Accurate Model of an SMT Processor (Modified from [ON+ 96]).... 33
FIGURE 4.2: Weighted IPC and Fairness Definitions.... 35
FIGURE 5.1: Impact of Disabling Speculative Execution for Superscalar Processors.... 39
FIGURE 5.2: Throughput IPC for 2-threaded Workloads with DISPEX, DISPEX+SF, and ADISPEX+SF.... 45
FIGURE 5.3: Throughput IPC for 3-threaded Workloads with DISPEX, DISPEX+SF, and ADISPEX+SF.... 46
FIGURE 5.4: Throughput IPC for 4-threaded Workloads with DISPEX, DISPEX+SF, and ADISPEX+SF.... 46
FIGURE 5.5: Fairness for 2-threaded Workloads with DISPEX, DISPEX+SF, and ADISPEX+SF.... 48
FIGURE 5.6: Fairness for 3-threaded Workloads with DISPEX, DISPEX+SF, and ADISPEX+SF.... 48
FIGURE 5.7: Fairness for 4-threaded Workloads with DISPEX, DISPEX+SF, and ADISPEX+SF.... 49
FIGURE 5.8: Performance of 8-threaded Workloads with DISPEX.... 52
FIGURE 5.9: Throughput IPC for 2-threaded Workloads with DYCOSPEX.... 56
FIGURE 5.10: Throughput IPC for 3-threaded Workloads with DYCOSPEX.... 57
FIGURE 5.11: Throughput IPC for 4-threaded Workloads with DYCOSPEX.... 57
FIGURE 5.12: Fairness for 2-threaded Workloads with DYCOSPEX.... 58
FIGURE 5.13: Fairness for 3-threaded Workloads with DYCOSPEX.... 59
FIGURE 5.14: Fairness for 4-threaded Workloads with DYCOSPEX.... 59

FIGURE 5.15: Throughput IPC for 2-threaded Workloads with COCOSPEX............ 61
FIGURE 5.16: Throughput IPC for 3-threaded Workloads with COCOSPEX............ 62
FIGURE 5.17: Throughput IPC for 4-threaded Workloads with COCOSPEX............ 62
FIGURE 5.18: Fairness for 2-threaded Workloads with COCOSPEX......................... 63
FIGURE 5.19: Fairness for 3-threaded Workloads with COCOSPEX......................... 64
FIGURE 5.20: Fairness for 4-threaded Workloads with COCOSPEX......................... 64
FIGURE 6.1: Single Thread Results for Disabled Speculative Scheduling................ 68
FIGURE 6.2: Pipeline Diagram for Load-Hit Speculation with Correct Hit Prediction.. 69
FIGURE 6.3: Pipeline Diagram for Load-Hit Speculation with Incorrect Hit Prediction.. 69
FIGURE 6.4: Execution Schedule Without Load-Hit Speculation............................ 69
FIGURE 6.5: Throughput IPC for 2-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays.............................. 71
FIGURE 6.6: Throughput IPC for 3-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays.............................. 72
FIGURE 6.7: Throughput IPC for 4-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays.............................. 73
FIGURE 6.8: Fairness for 2-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays.............................. 74
FIGURE 6.9: Fairness for 3-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays.............................. 75
FIGURE 6.10: Fairness for 4-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays.............................. 76
FIGURE 7.1: Pipeline Diagram with Back-to-Back Execution................................. 78
FIGURE 7.2: Pipeline Diagram with No Back-to-Back Execution........................... 79
FIGURE 7.3: Pipeline Diagram (2-way SMT) with No Back-to-Back Execution..... 79
FIGURE 7.4: Disabling Back-to-Back Execution on a Single Threaded Processor... 80
FIGURE 7.5: Throughput IPC for 2-threaded Workloads with Disabled Back-to-Back Execution.. 82
FIGURE 7.6: Throughput IPC for 3-threaded Workloads with Disabled Back-to-Back Execution.. 83
FIGURE 7.7: Throughput IPC for 4-threaded Workloads with Disabled Back-to-Back Execution.. 83
FIGURE 7.8: Fairness for 2-threaded Workloads with Disabled Back-to-Back Execution.. 84
FIGURE 7.9: Fairness for 3-threaded Workloads with Disabled Back-to-Back Execution.. 85
FIGURE 7.10: Fairness for 4-threaded Workloads with Disabled Back-to-Back Execution.. 85
FIGURE 7.11: Throughput IPC for 2-threaded Workloads for Various Selection Methods.. 89
FIGURE 7.12: Throughput IPC for 3-threaded Workloads for Various Selection Methods.. 90
FIGURE 7.13: Throughput IPC for 4-threaded Workloads for Various Selection Methods.. 91

FIGURE 7.14: Fairness for 2-threaded Workloads for Various Selection Methods..... 92
FIGURE 7.15: Fairness for 3-threaded Workloads for Various Selection Methods..... 93
FIGURE 7.16: Fairness for 4-threaded Workloads for Various Selection Methods..... 94

List of Tables

TABLE 4.1: Configuration of the 4-way SMT Machine... 36
TABLE 4.2: 2-Threaded Workloads... 36
TABLE 4.3: 3-Threaded Workloads... 37
TABLE 4.4: 4-Threaded Workloads... 37
TABLE 5.1: 8-Threaded Workloads... 53
TABLE 5.2: Summary of Results.. 65
TABLE 6.1: Summary of Disabled Speculative Scheduling on Superscalar............. 69
TABLE 6.2: Summary of Disabled Speculative Scheduling on SMT........................ 76
TABLE 7.1: Summary of Pipelining Scheduling Loops Over 2 Cycles on SMT...... 86
TABLE 7.2: Summary of Various Selection Methods on SMT................................ 95
TABLE 8.1: Summary of Disabling Speculative Execution..................................... 98
TABLE 8.2: Summary of Disabling Speculative Scheduling on SMT...................... 99
TABLE 8.3: Summary of Various Selection Methods on SMT................................ 100

Chapter 1

Introduction

Traditional architectural approaches for increasing microprocessor performance rely on the use of large, complex, highly-speculative out-of-order cores to extract Instruction-Level Parallelism (ILP) from single-threaded applications. In order to realize high performance, these designs employ a myriad of speculative techniques, ranging from branch prediction [S 81, PSR 92, M 93] to load-latency prediction [MB 02] and memory-dependence prediction [CE 98].

Branch prediction is used to predict the outcomes of conditional branches and speculatively execute the instructions along the predicted path without waiting for the actual branch outcome to be resolved. While it is essential for high performance in single-threaded execution environments, branch prediction incurs the overhead of having to deal with branch mispredictions, as the prediction accuracy is never perfect. In order to support branch mispredictions a complex logic needs to be incorporated to reconstruct the precise processor state and flush all instructions from the mispredicted path. Furthermore, instructions executed on the wrong-path simply waste the processor's resources, introducing overheads from both performance and power standpoints. In addition, the logic needed to implement the branch predictors themselves is often quite complicated, especially when very high prediction accuracy is important [J 03].

Another type of speculation used in recent superscalar designs is load-latency prediction [YE+ 99]. In order to avoid pipeline bubbles between the load and load-dependent instructions (i.e. reduce the load-to-use delay), the instructions dependent on the load are scheduled speculatively, without waiting for the cache hit/miss indication to be generated and instead assuming that the load hits into the L1 D-cache or using more sophisticated load hit/miss predictors [CR 00]. If the prediction is a miss, then the dependents of the load are not scheduled until the value is returned from the memory hierarchy. Otherwise, the instructions are scheduled just-in-time to pick up the value produced by the load from the bypass network. To illustrate the performance advantages that can be gained by using the load-hit speculation, consider the example shown in Figure 1.2. Corresponding pipeline diagram is depicted in Figure 1.1.

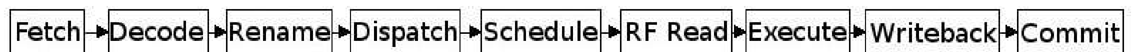


Figure 1.1: Pipeline Diagram

The importance of load-hit speculation is best described by the following examples. Figure 1.2 shows the execution schedule for a load and its dependent instructions without load-hit speculation, assuming the instruction hits into the L1 cache; if it misses delays occur with or without load-hit speculation.

Load Inst.	**W/S**	**RF**	**EX**	**WB**		
Dependent Inst.				**W/S**	**RF**	**EX**
Cycle:	**t**	**t+1**	**t+2**	**t+3**	**t+4**	**t+5**

Figure 1.2: Execution Schedule Without Load-Hit Speculation

As shown in Figure 1.2, a multi-cycle bubble results if the dependent instruction is forced to wait until it is determined if the load hits or misses into the L1 cache. Since most load instructions hit into the L1 cache [YE+ 99] it is worthwhile to speculatively schedule the dependent instruction to pick up the load's result just-in-time. When dependent instructions are scheduled in this manner, additional hardware is needed to handle the situation where the load is unable to execute due to a cache miss and the dependent instructions are unable to obtain the correct value, resulting in a replay. This means that resources were used to issue an instruction that cannot execute and could have been used for an executable instruction. These instructions fail execution and return to the issue queue to be selected again. For long load latencies these replayed instructions may be replayed multiple times while waiting for the load to be serviced. An intelligent replay scheme will only cause dependent instructions to replay, however a simple replay scheme will replay all instructions that follow the load in program order.

Instead of wasting resources replaying instructions a load-hit predictor can be implemented to make a prediction about the success of a load instruction hitting into the L1 cache instead of assuming that they all will. A basic predictor implementation can maintain hit/miss information about the most recent executions of a given load instruction in order to provide a basis for speculating on the same load instruction.

Adding a predictor can further reduce the replays that occur by scheduling the dependent instructions when the load misses into the L1 cache. Figure 1.2 also represents the pipeline diagram when the load instruction is not predicted to hit into the L1 cache

(although, larger bubbles are also possible). A correct prediction to hit into the L1 cache allows the execution of the load-dependent instructions to occur back-to-back without having to generate a replay. When the prediction is incorrect the load instruction and all of its dependent instructions need to be replayed once the load resolves generating a pipeline bubble of a non-deterministic latency.

Another important factor in sustaining high throughput of single-threaded workloads is to guarantee that the dependent instructions can be executed in the back-to-back cycles. This requirement often implies that the operations associated with the scheduling logic (namely, instruction wakeup and selection) must be implemented as an atomic operation. If these activities are pipelined, then either the ability to execute dependent instructions back-to-back is lost (hindering performance) or significant additional complexities have to be added throughout the pipeline to support such back-to-back execution.

The importance of back-to-back execution and the atomicity of the scheduling logic are best described by considering a code fragment. The following code fragment (Figure 1.3) will be evaluated with an atomic scheduler and a pipelined, two cycle, scheduler.

```
I1. ADD r1,r2,r3            /* r1 = r2 + r3 */
I2. ADD r4,r1,r5            /* r4 = r1 + r5 */
I3. ADD r6,r1,r4            /* r6 = r1 + r4 */
```

Figure 1.3: Code Example to Examine Back-to-Back Execution

This example encompasses the select-to-writeback portion of the pipeline. All of the instructions are assumed to be in the issue queue at cycle *t* with instruction I1 flagged as ready and selected. For simplicity, each of the stages is assumed to be one cycle (except pipelined wakeup/select), each of the instructions has an execution latency of one cycle

and the instructions access the register file after being selected.

I1	W/S	RF	EX	WB		
I2		W/S	RF	EX	WB	
I3			W/S	RF	EX	WB
Cycle:	t	t+1	t+2	t+3	t+4	t+5

Figure 1.4: Pipeline Diagram for Figure 1.3 with Atomic Scheduling

The ideal situation is when wakeup/select activities are atomic and it is represented by the pipeline diagram of Figure 1.4. The pipeline stages are described with the following notation: W/S is for the atomic wakeup/select, W is for wakeup, S is for select, RF is for accessing the register file, EX is for execution, WB is for writeback. As Figure 1.4 shows, I1 is selected at cycle *t* and can then access the register file at cycle *t+1*, execute at cycle *t+2*, and writeback at cycle *t+3*. I2 can be scheduled just-in-time to pick up the result from the execution of I1. Therefore, I2 is woken up and selected during cycle *t+1*. In this example, as I1 is selected it broadcasts that its result will be ready at cycle *t+2* allowing I2 to be woken-up during cycle *t+1*. After I2 is selected it will then broadcast that it will be done at cycle *t+3* and reads the register file during cycle *t+2* (obtaining the result from I1 via the bypass network) and executes during cycle *t+3*. I3 is selected during cycle *t+2*, and writes back at cycle *t+5*, thus completing the example code.

I1	W	S	RF	EX	WB				
I2			W	S	RF	EX	WB		
I3					W	S	RF	EX	WB
Cycle:	t	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8

Figure 1.5: Pipeline Diagram for Figure 1.3 with Pipelined Scheduling

The tight scheduling loop can be relaxed by pipelining wakeup and select over two cycles. Figure 1.5 shows the timing diagram for this case where I1 is woken up at cycle *t*

and selected at cycle *t+1*. When I1 is selected it broadcasts that its result will be ready during cycle *t+3*. I1 accesses the register file at cycle *t+2*, executes during cycle *t+3*, and writes back at cycle *t+4*. I1's broadcast during cycle *t+1*, allows I2 to be woken up next cycle and be selected during cycle *t+3*. During cycle *t+3* I2 broadcasts that its results will be ready for cycle *t+5*. I2 will access the register file during cycle *t+4*, execute during *t+5*, and write back during *t+6*. I3 will be able to wake up during cycle *t+4*, select and broadcast during cycle *t+5*, read the register file at cycle *t+6*, execute at cycle *t+7*, and write back at cycle *t+8*. This example demonstrates that there is a pipeline bubble that occurs between the execution of each of these instructions. No longer are these instruction able to execute back-to-back but are now separated by a one cycle pipeline bubble. Compared to atomic wakeup/select this takes three additional cycles to complete. This one cycle bubble will affect all instructions that execute within a single cycle. Since this is a majority of the instruction set it is obvious that non-atomic wakeup and select can cost additional cycles per instruction.

Despite these significant complexities, embodied in highly-speculative techniques and tight pipeline loops, the ILP-extracting techniques have reached the point of diminishing returns, where the design complexities and additional power consumption simply outweigh the small performance advantages. This is mainly due to the growing processor-memory gap [HP 97], where the processor waits a very large number of cycles for every memory access (i.e. a miss into the L2 cache). Figure 1.6 shows these trends.

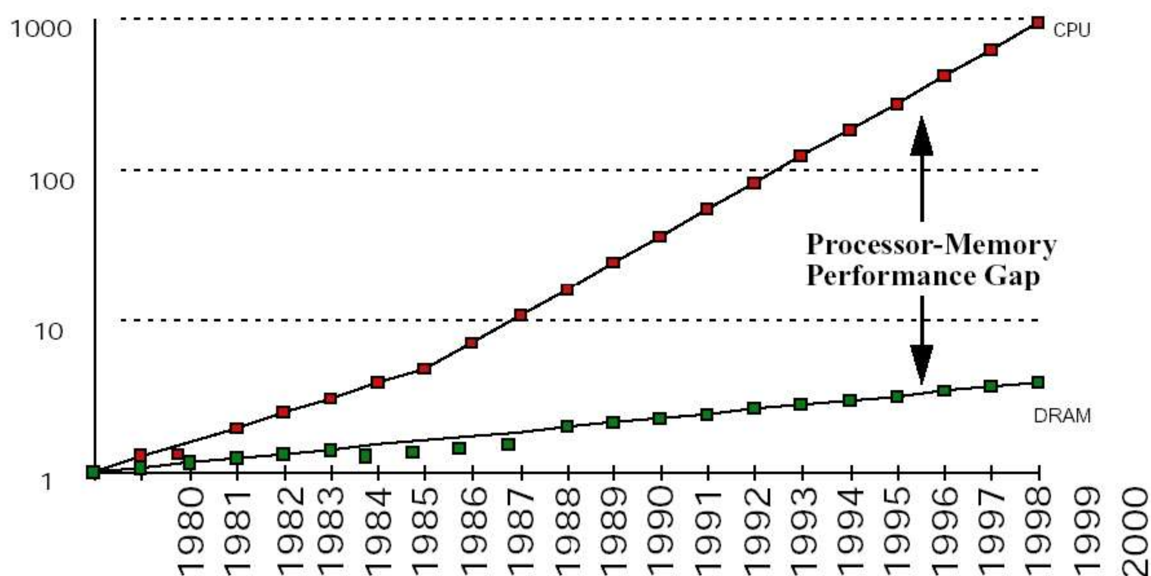


Figure 1.6: Processor-Memory Gap [HP 97]

Currently, memory latencies are exceeding 300 cycles [TT 03] and stalling the processor activities for such a long time limits the performance to a point where further advanced in speculation are simply very difficult to justify. Unless some exotic and complicated techniques such as Run ahead execution [MKP 06], value prediction [LWS 96] or checkpoint-based processing [FR+ 02] are used, the reorder buffer quickly fills up impeding further exploitation of the ILP. Therefore, the focus of the research community has recently shifted to exploiting Thread-Level Parallelism (TLP) in addition to, or instead of, the ILP.

The TLP can generally be harvested through two design philosophies: Chip Multiprocessing (CMP) and Simultaneous Multithreading (SMT). In a CMP design, several processors are placed on the same die, so that multiple programs can execute concurrently, each having one processor in its full disposal. The processors in a CMP

usually share some lower levels of the memory hierarchy (such as the L2 cache and below). In SMT designs, in contrast, multiple programs execute simultaneously on the *same* processor, and they *dynamically share* the processor resources. Compared to a traditional superscalar machine, SMT processor better utilizes the available processor's resources and therefore can support higher overall instruction throughout, as instructions are selected for the execution from multiple streams [TEL 95].

In this thesis, we investigate multithreaded (SMT) processors and perform a comprehensive quantification of the impact that branch speculation, load-hit prediction, and scheduler design has on the performance of a multithreaded machine compared to a traditional superscalar.

1.1 Goals and Contributions

While in general, an SMT processor can use all existing features of the underlying superscalar, the presence of inherent TLP may provide some opportunities to simplify the logic, avoid some forms of speculation or relax some tight pipeline loops.

The goal of this thesis is to investigate the impact of disabling or limiting the branch and load-hit speculation, as well as the impact of pipelining the scheduling logic on the performance of SMT processors.

To begin it is required to examine the synergy of speculative execution with multithreading. If sufficient TLP exists, then disabling the speculative execution on SMT

can, at least in theory, result in the allocation of resources to only non-speculative instructions thus possibly even increasing the performance. Using 2, 3 and 4-threaded mixes of SPEC 2000 benchmarks, quantification can be made of the impact of completely disallowing speculative execution as well as only resorting to the speculative execution when no non-speculative instruction from any thread is available for fetching.

Then the impact of disabling load-hit speculation on SMT is examined. The motivation is that the bubbles created as a result of such restriction are likely to be filled by the load-independent instructions from other threads on SMT [TEL 95], thus possibly mitigating the performance impact.

Lastly, the impact of pipelining the instruction scheduling logic into separate wakeup and selection stages is quantified. Just as with load-hit speculation, if the pipeline bubbles are filled with the instructions from other threads, then the impact on the performance can be negligible. Various instruction selection schemes specifically targeted towards this goal are proposed and analyzed.

1.2 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 presents a detailed description of the SMT architecture. Chapter 3 discusses related works concerning the exploitation of ILP in a single threaded machine and performance enhancing techniques that can be applied to SMT. Chapter 4 describes the simulation environment and the metrics used to evaluate all of the techniques described in this thesis. Chapter 5 quantifies the impact of

disabling branch prediction on SMT machines and explores techniques to alleviate the resulting performance challenges. Chapter 6 evaluates the importance of load-hit prediction on SMT by disabling L1 cache speculation. Chapter 7 examines the penalty for removing back-to-back execution and considers how its removal can enable pipelining of the selection logic. Various selection policies are examined to mitigate the penalty resulting from the removal of back-to-back execution. Lastly, Chapter 8 formalizes our conclusions and outlines directions for future research.

Chapter 2

Datapath Organization of Simultaneously Multithreaded Processors

This chapter examines the datapath of a typical SMT machine as shown in Figure 2.1. Many aspects of the SMT machine are similar to that of a typical superscalar machine with variations that accommodate the handling of multiple threads. All of the pipeline stages will be presented with a focus on the additional features present in SMT compared to superscalar. Additional sections will discuss the instruction scheduling logic and branch predictor that are used by the default SMT machine.

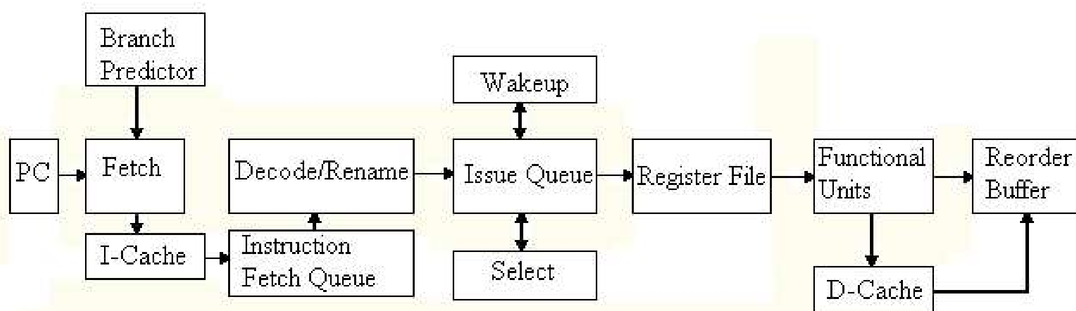


Figure 2.1: SMT Pipeline Diagram

2.1 Organization of a Basic SMT Datapath

The basic SMT datapath is pipelined over several stages that act on in-flight instructions during each cycle. These instructions are obtained by the fetch stage from the instruction caches in program order. Instructions are fetched using the ICOUNT technique (see Section 2.3) and can be fetched from any of the available threads. To avoid waiting for

branch instructions to resolve, and ensure a continuous flow of instructions, branch prediction is used to speculate the direction of the branch. When a branch instruction is fetched the branch predictor uses its prediction to change the program counter (PC) to the speculated address. When the prediction is incorrect the precise state of the thread must be recovered by flushing the pipeline of incorrectly fetched instructions and restoring the PC to the correct value.

Once instructions are fetched, they are placed into private instruction fetch queues while they wait to enter the decode/rename stages. The decode portion determines the source and destination registers and renames them from their private architectural register value to a physical register from a common pool.

Register renaming removes write-after-read and write-after-write data hazards. Write-after-write data hazards occur when an earlier write overwrites a successive write to the same register causing the wrong results. Figure 2.2 provides a code snippet to demonstrate this. In an out-of-order processor it is possible that I2 completes first, writing its result into R1, then I1 completes and overwrites the value in R1. Clearly, the value in R1 is not the correct value to use for instruction I3.

```
I1. ADD r1,r2,r3              /* r1 = r2 + r3 */
I2. ADD r1,r6,r5              /* r1 = r6 + r5 */
I3. MUL r9,r1,r1              /* r9 = r1 + r1 */
```

Figure 2.2: Code Example to Examine Write-after-Write Data Hazard

Write-after-read data hazards are a result of a write occurring before the appropriate read to the same register (See Figure 2.3). If I2 completes before I1 reads R1 then the value in

R1 is overwritten and I1 will not be able to read the correct data.

```
I1. ADD r4,r2,r1              /* r4 = r2 + r1 */
I2. ADD r1,r6,r5              /* r1 = r6 + r5 */
```

Figure 2.3: Code Example to Examine Write-after-Read Data Hazard

These hazards are removed by mapping the architectural registers used by each thread to the pool of physical registers shared among all of the threads. This can be observed by considering Figure 2.4 which shows the application of register renaming to both of the data hazards discussed previously.

```
I1. ADD r1,r2,r3      ->      ADD p1,p2,p3
I2. ADD r1,r6,r5      ->      ADD p4,p6,p5
I3. MUL r9,r1,r1      ->      MUL p9,p4,p4
```

(a) Register Rename for Write-after-Write Data Hazard

```
I1. ADD r4,r2,r1      ->      ADD p4,p2,p1
I2. ADD r1,r6,r5      ->      ADD p3,p6,p5
```

(b) Register Rename for Write-after-Read Data Hazard

Figure 2.4: Application of Register Rename to Data Hazards

For both Figure 2.4a and 2.4b out-of-order execution no longer generates a problem. Write-after-write hazards are avoided because I3 no longer has a false dependency on I1 and I1's result has no impact on the source operands of I3. Write-after-read hazards are also avoided, I2 can no longer overwrite the source operand of I1. This is all accomplished by maintaining a list of architectural registers for each thread and attaching a tag to a physical register for each of these architectural registers. These tags indicate where the value that reflects the architectural register exists within the physical register file. When an instruction is renamed the source operands are changed to the tag for the matching architectural register. Destination registers receive a new unallocated physical register and update the appropriate architectural register tag. This requires that the instructions are renamed in-order but does not restrict the execution of them, ensuring

correct execution.

After instructions are renamed they can be dispatched into the issue queue and reorder buffer (ROB) in program order. From the issue queue instructions can be issued out-of-order as long as all of their dependencies can be met before/at execution. Issued instructions access the physical register file to read their source operands and then activate the appropriate functional unit (FU) in order to begin execution.

Multiple FU types exist to perform various operations. These FUs are separated into two groups; one for integer operations and the other for floating point operations. FUs are used for mathematical, logical, shifting, load and store operations and have varying latencies that the scheduling logic is made aware of. With the exception of load instructions, these latencies are the basis for the scheduling logic to create an efficient flow of instructions; load latency is non-deterministic.

After the FUs complete execution the instructions enter writeback where their results, when applicable, are written back to the physical register file. Then, if the instruction is at the head of the ROB the instruction can be committed which updates the architectural register file and removes the ROB entry, allowing the next ROB to be considered for commitment. Instructions are allowed to be executed out-of-order but they can not update the architectural state out-of-order. The use of the ROB to ensure in-order updates to the architectural state permits precise state recovery in case of branch misprediction or exceptions.

2.2 Scheduler Design

Once an instruction is renamed and dispatched it is placed into the issue queue and can be scheduled for execution. The scheduler examines instructions that reside in the issue queue and select those with resolved dependencies. This is accomplished by checking the valid bit of the source physical registers which indicates that a result has been written to the register.

Only N instructions (from possibly multiple threads), determined by the issue-width of the processor, can be selected for execution each cycle which is also limited by the number of appropriate functional units that are available. In order to improve performance, instruction selection can occur when source operands are not yet ready but can be received just-in-time from the register bypass network allowing avoidance of pipeline bubbles that result from stalling until a source register has committed to the architectural state.

As a critical pipeline component, the scheduler is responsible for providing an optimal flow of instructions through the pipeline. However, current trends of increasing processor frequency place additional strain on the scheduler which can only be reduced by pipelining the scheduler or creating a more efficient scheduling method that can be expected to maintain single cycle completion while frequencies continue to increase. Pipelining the scheduler removes the atomicity that exists when the scheduler issues instructions each cycle, instead back-to-back execution would no longer be possible with

the traditional scheduler which forces a pipeline bubble between all dependent instructions proportional to the number of cycles needed by the scheduler [PJS 97]. Without back-to-back execution instructional-level parallelism is reduced degrading performance on a single-threaded pipeline that an SMT machine can hide.

2.3 Instruction Fetcher Design

All correct-path fetched instructions ultimately become a part of the issue queue and require resources as they enter the pipeline. When instructions enter the pipeline and stall soon thereafter issue queue clog [EA 03] is created; wasting critical resources that other threads can use to make progress. In a single-threaded machine this is inevitable and out-of-order execution is used to improve performance by exploiting instruction-level parallelism. A sufficiently long cache-miss can cause the entire pipeline to stall wasting hundreds to thousands of cycles. On an SMT machine the ability to exploit thread-level parallelism reduces the effect of cache-misses on the pipeline.

The traditional fetching logic used on SMT machines, ICOUNT, takes various queue usages into consideration when deciding which thread, or threads, to fetch from. ICOUNT maintains a counter for each thread that keeps track of the number of instructions in-flight and not-yet-issued. Threads with smaller values maintained in their counter are given priority compared to other threads. This works under the assumption that threads with a smaller amount of instructions in-flight or not-yet-issued are actively processing instructions and not wasting issue queue resources. ICOUNT also improves fairness by ensuring that no single thread dominates issue queue usage and ensures that

the various queues are filled with instructions that maximize thread-level parallelism. This is by no means perfect; critical resources must be wasted in order to detect and act on stalls and instructions are always fetched even if all threads are stalled. ICOUNT does not take any other information into account being unable to benefit from knowledge of load-hit misprediction or branch misprediction. Techniques described in future chapters will explain some modifications that can be made to ICOUNT in order to exploit such information.

2.4 Branch Prediction

In order to ensure that the processor has a suitable amount of instructions to execute a continuous flow of instructions need to be fetched from all available sources. Due to conditional branch instructions the fetching logic cannot simply fetch sequential instructions at its own discretion due to the inability to know which instructions are on the correct path. One option (explored in chapter 5) is to wait until the conditional branch instruction resolves and continue fetching. In a simplescalar environment it has been shown that stalling is not effective and predicting the direction of the branch instruction leads to effective instruction-level parallelism. Therefore, dedicated branch prediction logic can predict the direction of branch instructions at fetch time and update the program counter to reflect the predicted control path [PSR 92, M 93, EP+ 98].

The branch prediction logic relies on a feedback method based on comparing the prediction to the actual branch direction derived from execution to continually improve the accuracy of predictions. These updates are made when the conditional branch instruction is committed or can be done speculatively.

Branch prediction requires that precise state recovery can be supported in case of misprediction. Mispredictions result in fetching of wrong-path instructions that occupy critical processor resources that must be removed without having an effect on the architectural state. Misprediction is more expensive than stalling the processor to wait for a result but this penalty is considered to be worthwhile considering high accuracies achieved by modern branch predictors.

2.4.1 Branch Predictors

Branch prediction can be implemented using a variety of different techniques in order to speculate the result of a conditional branch instruction. A bimodal branch predictor uses a set of two-bit saturating counters to represent the branch direction history for a small group of instructions, indexed by their least significant bits. When a branch is taken the appropriate counter is incremented by 1 and when it is not taken it is decremented by 1. When the branch is fetched a prediction is made based on the value of the counter, if the most-significant bit of the counter is set then the branch is predicted to be taken.

The bimodal approach can be modified to take high-level coding constructs into consideration by using a global history for the set of most recent branches. This

technique, gselect, permits the usage of the bimodal predictor to scale with large table sizes. Further modifications address the the counters using XOR instead of taking the least significant bits; this is called gshare. Gshare reduces the need for large table sizes and is the default predictor used throughout this work.

Sometimes, it pays off to use multiple predictors and a method to select among them. A two-level predictor implements two standard predictors and uses a choice predictor (which is just a two-bit saturating counter) to choose between them. This is done to attempt to gain the benefits of multiple predictors since predictors may be optimal for different code phases.

Just like the selection stage, the branch predictor is subject to strain due to faster clock frequencies and shorter cycle time. Under these stresses a fast, and simpler, predictor can be used to make a quick prediction with a slower, more comprehensive, predictor used to verify the initial prediction. The slow predictor can override the initial prediction and restore the precise state with less waste of critical resources than a traditional misprediction.

Chapter 3

Related Work

This section will discuss the previous works that are related to the atomicity of the scheduler, load-hit speculation, branch prediction, and fetch policies of SMT. The prior work demonstrates that recent trends have been considering exotic methods to extract ILP from the single-threaded machine. TLP can increase performance in lieu of ILP and could allow much less aggressive forms of speculation to be used.

3.1 Execution Schedulers

Scheduling of instructions determines what instructions are eligible for execution and plays a critical role in the performance of the processor making the dynamic instruction scheduler a critical pipeline resource both in superscalar and SMT processors. In this section two approaches for building larger schedulers, without commensurately increasing the clock cycle time are considered: pipelined scheduling logic [SBP 00], and a selection-free implementation [BSP 01].

3.1.1 Pipelined Scheduling with Speculative Wakeup

If scheduler access is pipelined, instructions may still be able to wakeup for back-to-back execution. This can be done by speculatively determining when to wake up an instruction based on the information about their grandparents (Figure 3.1) [SBP 00].

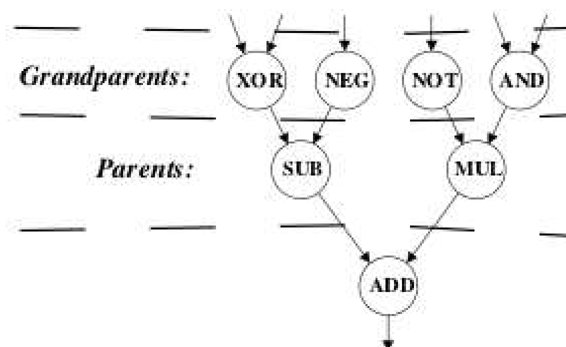


Figure 3.1: Example Data Flow Graph [SBP 00]

Figure 3.1 shows that ADD consumes the results of SUB and MUL, then SUB and MUL are the parents of ADD and ADD is the child of SUB and MUL.

When an instruction's grandparents are selected it is likely that the parents will be selected in the following cycle. For instructions of varying known latencies the speculation can be adjusted accordingly. In this manner scheduling can be based on the grandparents of an instruction; this allows scheduling two cycles in advance and therefore covers up the problem of pipelining the scheduling logic across two cycles. However, this is a speculative method. Even if the grandparents of an instruction complete in one cycle it does not guarantee that the parents are scheduled; requiring logic to replay instructions that were scheduled incorrectly.

The number of incorrectly scheduled instructions is influenced by the amount of contention for the functional units. In an SMT environment there are many instructions in-flight and contention for the functional units may be very high due to the availability of instructions. However, SMT may reduce the need for back-to-back execution by providing more instructions to work with (see Chapter 6).

3.1.2 Select-Free Instruction Scheduling Logic

The approach proposed by Brown, Stark and Patt [BSP 01] removes the selection logic from the critical path by assuming that instructions that are woken up can be selected immediately for execution, reducing the need for selection logic and eliminating existing stresses on the wakeup/select stage. This would allow more complex wakeup implementations to be pursued.

This method is justified by showing that 92% of instructions that are woken up are immediately selected in the baseline case. When applying this method, the remaining instructions are assumed to always be ready and a scoreboard is used to keep track of available functional units and detecting a data hazard called a *pileup*. When there are not enough functional units available the instruction is said to have had a *collision*. This leads to instructions dependent on the instruction that had a *collision* to *pileup* and not be executable. *Collision* and *pileup* lead to scheduling penalties that decrease performance. This requires the modification of the datapath as shown in Figure 3.2.

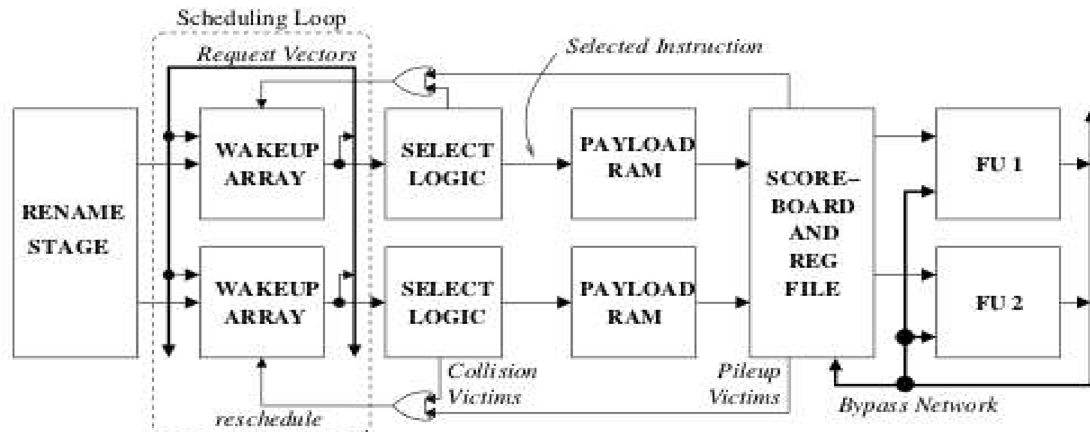


Figure 3.2: Execution Core of a Processor with Select-Free Scheduling [BSP 01]

This solution does not take into consideration the problem of decreasing cycle times. Pipelining of the scoreboard logic will increase scheduling penalties by increasing the number of stages between wakeup and the scoreboard.

In an SMT environment there is an expectation to see a large number of *collisions* due to the excess of available instructions. *Pileup* is less likely to happen as long as the wakeup of instructions is relatively fair since wakeup occurs from multiple threads and are less likely to wake up an instruction that is dependent on a *collision* instruction. However, this would require changes to the scheduling method to avoid *collisions*. Perhaps a queue or stalling of prior stages can alleviate concerns over *collision* instructions.

3.2 Load Latency Prediction

In order to maximize the use of the processor resources there needs to be as many in-flight instructions, that are moving through the pipeline at a sufficient rate, as possible. For most instructions the latency of the instruction can be evaluated and taken into

consideration for dependent instructions. This is due to most instructions having a deterministic latency.

However, load instructions do not conform to this standard because they have a variable non-deterministic latency. Ideally, the desired value is in a nearby cache but when a load instruction misses into these caches, often referred to as a L1 or L2 cache miss, the load becomes a bottleneck in the pipeline. It cannot complete execution because it is waiting to be serviced by a later cache (or possibly the disk), and its dependents are waiting on the load itself. Without detection, instructions can continue to be fetched from the context containing the load and take up critical resources due to in-order commitment requirements.

The difficulty here is in the inability to know if a load will miss or hit into the L1 cache. If known, the scheduler could schedule around it (in-order commitment limits this in a single thread machine) by fetching instructions from other contexts or executing out-of-order. A number of speculative approaches will be discussed in this section that deal with this problem; these include prediction techniques and additional hardware to mitigate the in-order commit problem. In Chapter 6, a conservative method to deal with load latency speculation will be evaluated on SMT to determine how well additional thread context support can hide latency.

3.2.1 Managing Load-Related Scheduling Issues on SMT

Tullsen and Brown [TB 01] discuss various methods to handle load latency issues with

regard to SMT and explain that when a context is stalled SMT continues to fetch instructions and maintain resources for that thread. This clogs up vital shared resources such as the issue queue. They demonstrate this by showing that the issue queue is only occupied 62% when no misses are present but that just one cache miss brings this to 97%. This demonstrates the disastrous effect of starving other threads that a cache miss can have on the shared issue queue.

The first step to solving this problem is to discover when long latencies have occurred. When a load misses the L1 cache, exploiting TLP can cover up the latency that is required to access the L2 cache. Missing the L2 cache causes a much greater delay that cannot be hidden with TLP therefore there is reliance on methods that detect an L2 cache miss and can suggest various ways to handle the latency.

One method described by Tullsen and Brown [TB 01] relies on evaluating how much latency a load has incurred such that at a particular threshold it is obvious that it missed the L2 cache. When determining this threshold contention between the contexts as well as the time to access the L2 cache must be taken into consideration. An easier method, but requiring more hardware, allows the L2 cache to signal that a miss has occurred.

Regardless of what method is used to determine if an L2 cache miss has occurred a method is needed to prevent clogging of the shared resources and, ideally, retrieve them. Some of the techniques described by Tullsen and Brown [TB 01] involve flushing the pipeline at various points after a load missing into the L2 cache is detected. They make an

important conclusion about when the flushing should occur based on the number of contexts that the SMT core can handle; less contexts equates to flushing closer to the load, in other cases flushing before the loaded data would be used was best.

The problem with flushing is that instructions may be removed from the pipeline unnecessarily. To handle this, Tullsen and Brown [TB 01] also present stalling as an option to handle L2 cache misses and use TLP to cover up performance losses; this prevents further clog of the critical resources. Their results indicate that stalling is not an effective solution since TLP cannot compensate for such long latencies.

3.2.2 Load-Hit Misspeculation Techniques

Within the processor pipeline instructions are often scheduled several cycles before they can execute requiring logic to recover when a scheduled instruction cannot execute. When a load is scheduled speculatively it can wake up its dependent instructions so they execute just-in-time to pick up the results of the load from the bypass network. If the load misses into the L1 cache it, as well as its dependents, must be replayed in order to ensure proper pipeline function.

Various replay implementations are described by Kim and Lipasti [KL 04] that can be implemented with varying hardware complexities. Issue-Queue-Based Replay requires that all loads wait in the issue queue until they can be serviced. Dependent instructions enter the issue queue while the load is waiting to be serviced and create issue queue clog [EA 03]. Refetching Replay flushes all of the instructions for a misspeculated thread from

the pipeline. This has the benefit of being simple to implement but harmful to performance due to flushing of non-dependent instructions that must be refetched. Non-Selective Replay invalidates all scheduled instructions for that thread and forces them to be rescheduled. This repeats until the load instruction can complete execution causing many non-dependent instructions to be replayed. Selective Replay increases complexity by determining which instructions are dependent on a violating load and rescheduling only those dependent instructions. One implementation can be accomplished by broadcasting kill tags to dependent instructions and having them propagate the broadcasts with their own dependents. Another method uses poison bits [GA+ 05] that are sent over the bypass network, and put into the register file, if a load misses into the L1 cache. When dependent instructions read from the register file, or receive a result from the bypass network, the poison bit causes the instruction to be replayed.

3.2.3 Additional Resources to Combat Load Latency Misprediction Costs

The behavior of various pipeline structures can be changed in order to reduce the bottlenecks that are caused by cache misses. One method is to create a private issue queue that is used when a load experiences a cache miss to store the load and instructions dependent on the load. This relieves pressure on the issue queue while waiting for the load to be serviced.

A method examined by Morancho, Llabería and Olivé [MLO 01] uses a recovery buffer (Figure 3.3) that can be used along side the issue queue. This buffer takes appropriate loads, stores and dependents that are issued and places them into this buffer in program

order. This reduces the pressure on the issue queue by getting the potentially problematic instructions out of the issue queue.

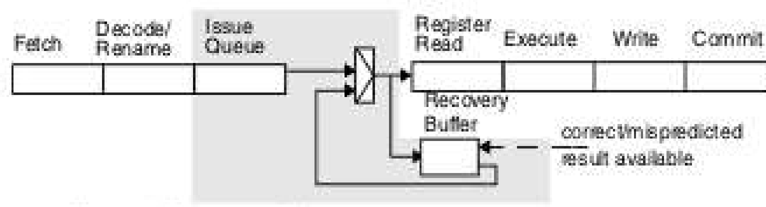


Figure 3.3: Placement of the Recovery Buffer [MLO 01]

When a misprediction occurs instructions can be issued from the recovery buffer instead of the issue queue allowing a reduction of the complexity of the issue queue and supporting re-issuing via a more optimized structure. This reduces the problem of long latency instructions taking up many critical resources since the recovery buffer can absorb these instructions and allow non-dependent instructions into the issue queue.

3.3 Branch Prediction

When applying a branch predictor designed for superscalar to SMT the hardware cost of the implementation needs to be taken into account. Full replication of the predictor for each context comes at a high cost and partial replication may not produce adequate results. Ramsay, Feucht and Lipasti [RFL 03] examined four methods of implementation the predictor configuration; Shared history shared predictor, separate history separate predictor, shared history separate predictor, and separate history shared predictor.

Of the predictor configurations a separate history using a shared predictor was shown to achieve results almost identical to full replication at a greatly reduced hardware cost. This is because a separate history for each context provides enough protection from the contexts interfering with each other by overwriting branch behavior history.

These results were nearly the same regardless of the actual predictor used. This is an important observation as it indicates that various branch predictors do not appear to perform significantly better than each other on an SMT core.

3.3.1 Current Trends: Pipelined Branch Predictors

In the previous subsection it is shown that various branch predictors have similar performance on an SMT core; leading to reduced motivation to find more efficient or better predictors. Current microprocessor trends are expected to force reevaluation of the complexity of existing branch predictors as reduced cycle times may require the pipelining of the branch predictor logic. This would result in a bottleneck for the fetcher since it would not be able to generate predictions without a pipeline bubble impacting performance.

There has been research in branch predictors that can be pipelined without impacting the fetcher's ability to use predictions immediately. A Hierarchical branch predictor uses a small, quick predictor that makes a prediction in one cycle and a complex predictor that verifies the prediction that takes multiple cycles. Jiménez [J 03] explains that if the complex predictor disagrees with the original prediction that the processor would have to rollback as if there was a branch misprediction, although less work was wasted.

Jiménez [J 03] proposes modifying the gshare predictor so that larger implementations of it can still have an effective prediction time of one cycle; called gshare.fast. This is accomplished by allowing gshare to become more complicated while providing a pattern history table (PHT) which is a small set of potential upcoming branches that can be quickly accessed (Figure 3.4). If future instructions can be brought just-in-time into the PHT then predictions can be made with one cycle latency while having access to a larger amount of branch history.

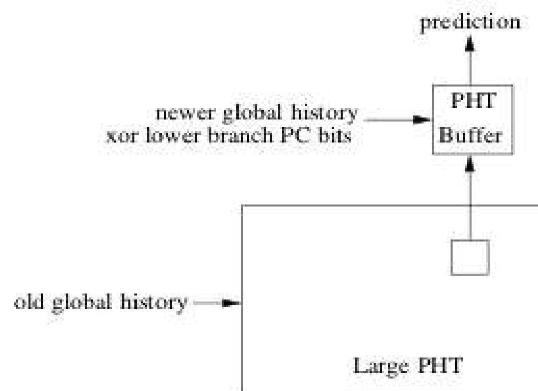


Figure 3.4: Separation of the gshare Predictor [J 03]

Considering the results from Ramsay, Feucht and Lipasti [RFL 03] it appears that gshare.fast can be effectively applied to an SMT core by adding additional PHTs as

separate histories for each context. Since these PHTs are small in size and the predictor can be pipelined effectively this method should be scalable as performance demands increase.

3.4 Fetch Policies

The mix of instructions that enter the pipeline is directly controlled by the fetching policy implemented. On a superscalar machine there is only one source for instructions so fetching is trivial unlike on an SMT machine where competing threads all vie for fetch bandwidth. The fetching logic for SMT needs to fetch from all threads in order to provide an optimal mix of instructions that are representative and traverse the pipeline efficiently. To accomplish this, Tullsen, et al, [TE+ 96] describe ICOUNT; a fetching policy that ensures a representative mix of instructions from all threads while preventing the clogging of critical resources by taking the occupancy of the decode, rename and issue queues into account.

ICOUNT lacks the ability to deal with excessive load-latency resulting in the reduction of issue queue resources. This has led to several optimizations being suggested to handle long-latency loads in an SMT. STALL [TB 01] disabled fetching from a thread if an L2 cache miss is detected. FLUSH [TB 01] not only disables fetching from a thread that experienced an L2 cache miss but also flushes all of that thread's instructions from the issue queue allowing other threads to use those IQ resources. FLUSH++ [CF+ 03] provides a balance between the STALL and FLUSH policies and uses a load-hit predictor to reduce the penalties due to flushing or useful instructions from the pipeline. The load-

hit predictor speculates if the thread will generate an L2 cache miss and will guide the fetching logic to select from threads that are expected to hit into either the L1 or L2 caches. Threads that are expected to generate an L2 cache miss can either be flushed, if IQ resources are needed, or stalled while the waiting for the load to resolve.

Instead of relying directly on L2 cache miss detection, the fetching policy can be guided by resource allocation techniques that dynamically adjust to the needs of the in-flight threads. DCRA [CR+ 04] permits fine grained control over all critical processor resources by classifying threads based on outstanding L1 cache misses and per thread demands on various resources. Threads without outstanding cache misses tend to move their instructions through the pipeline quickly using few resources and are given priority. Fine grained control of resources is achieved by monitoring resource needs of each thread. If a thread accessed a resource within a threshold of cycles it is given a share that resource, otherwise it forfeits its share to the other threads. This resource management technique allows threads that do not need resources to provide additional support to other threads.

Hill-Climbing [CY 06] takes IPC performance into consideration when determining how to distribute resources to the threads. Periodically, the IPC performance of the processor is evaluated and stored in a circular array called *perf* then a *Trial Thread* is selected, in a round robin fashion, to receive additional resources (*delta*) from the other threads. *Perf* is examined after every N performance evaluations in order to determine which thread is performing best giving that thread *delta* resources taken from the other threads.

Chapter 4

Simulation Methodology

To evaluate the ideas proposed in this thesis, M-Sim [SPG 06] - a cycle-accurate simulator of a multithreaded processor - is used. M-Sim is built on top of the SimpleScalar 3.0d simulator [BA 97] and significantly extends SimpleScalar even in a single-threaded execution environment in the following major ways: 1) It provides separate and accurate models for all key datapath components, including the issue queue, the reorder buffer, the register file and the register renaming table; 2) It explicitly models register renaming process; 3) It provides an accurate model of the schedule-to-execute part of the pipeline, support load-hit speculation and models various misprediction recovery schemes. The entire simulation framework is shown in Figure 4.1.

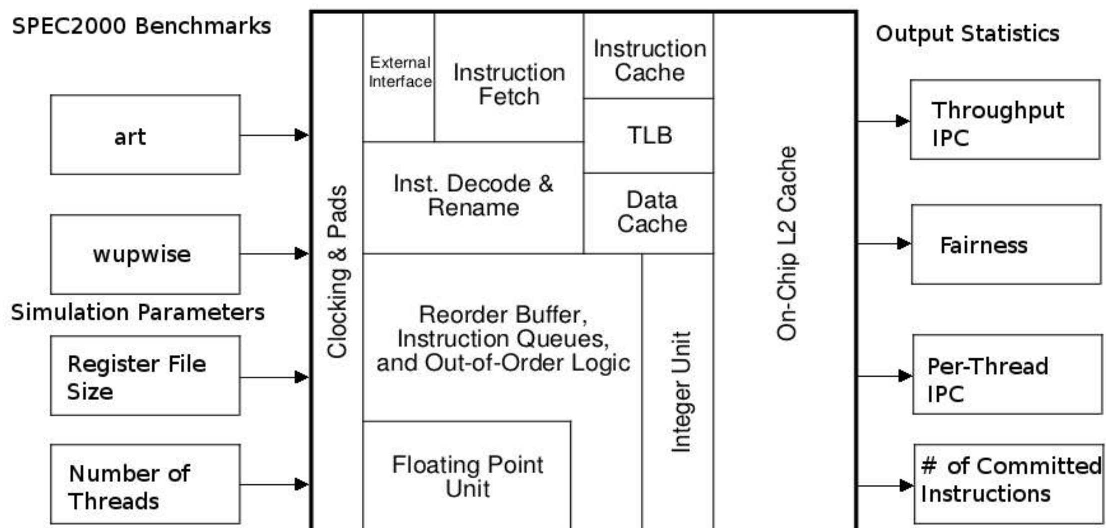


Figure 4.1: Cycle-Accurate Model of an SMT Processor (Modified from [ON+ 96])

These experiments used the full set of SPEC 2000 [H 00] benchmarks. The pre-compiled Alpha AXP binaries were obtained from the SimpleScalar website. (www.simplescalar.com). For each test, the first 100 million instructions are fast forwarded and simulation continues until the next 100 million instructions commit. In a multithreaded environment, simulations are stopped when 100 million instructions from at least one thread had committed.

To evaluate the performance of the multithreaded workloads, several metrics were used. The first metric is the throughput IPC, which is the cumulative IPC of all simultaneously executing threads. However, if a technique that is being evaluated favors the high-IPC threads, then the overall throughout can increase at the expense of hindering the progress of low-IPC threads. To address this issue, a different metric, called “fairness” (or harmonic mean of weighted IPCs) was proposed [LGF 01]. This takes into account the individual thread performance (basically, how much slower each thread is in a multithreading execution mode compared to its execution in the single-threaded environment). Fairness is computed by getting a weighted IPC value for each context in the multithreaded case; this is the per context IPC for a particular context divided by its IPC in a single threaded case. Then, take the harmonic mean of the weighted IPC values (Figure 4.2). The closer this value is to 1, the more balanced the execution scheduler is and less thread starvation is encountered. Realistically, a higher value is desired, but values close to 1 are not expected.

Weighted IPC

$$Weighted\ IPC = \frac{Per\ Thread\ IPC}{Single\ Thread\ IPC}$$

Fairness

$$Har_{mean} \sum_{1}^{n} Weighted\ IPC$$

Figure 4.2: Weighted IPC and Fairness Definitions

All of the results are evaluated in terms of fairness and throughput IPC that are obtained by simulating a multithreaded machine with the proposed modification and comparing these metrics against those obtained by simulating the baseline SMT machine. The processor configuration used is shown in Table 4.1, it applies for all tests, regardless of the number of thread contexts used.

Each of the considered benchmarks is executed in single threaded mode and sorted according to their commit IPCs into three groups: high-IPC benchmarks (IPC greater than 3.5), low-IPC benchmarks (IPC less than 2.5), and medium-IPC benchmarks (IPC between 2.5 and 3.5). Then randomly generated mixes of 2-threaded, 3-threaded and 4-threaded (Tables 4.2, 4.3, 4.4 respectively) workloads are comprised of the benchmarks that represent different combinations of these groups. Floating point versus integer benchmarks are not taken into consideration within groupings. In total, 6 2-threaded workloads, 10 3-threaded workloads and 9 4-threaded workloads are used.

Parameter	Configuration
Machine Width	8-wide fetch, 8-wide issue, 8-wide commit
Window Size	128 entry ROB, 64 entry IQ, 48 entry LSQ
Function Units and Latency (total/issue)	8 Int Add (1/1), 4 Int Mult/ (3/1) Div (20/19), 4 Load/Store (2/1), 8 FP Add (2/2), 4 FP Mult (4/1)/ Div (12/12)/ Sqrt (24/24)
Physical Registers	512 integer, 512 floating point registers
L1 I-Cache	32KB, 2-way set-associative, 32 byte line, 1 cycle hit time
L1 D-Cache	64KB, 4-way set-associative, 32 byte line, 2 cycle hit time
L2 Cache	Unified: 512KB, 8-way set-associative, 12 cycle hit time
BTB	2048 sets, 2-way set associative
Branch Predictor	gshare: 4k entries, 10-bit global history per thread
Memory	64 bit wide, 300 cycle first chunk access, 2 cycle interchunk access
Load-Hit Predictor	2-bit bimodal: 1k entries, 8-bit global history per thread.
TLB	128 entry (I), 64 entry (D), fully associative, 30 cycle miss latency
Fetch-Rename Delay	4 cycles
Rename-Dispatch Delay	1 cycle

Table 4.1: Configuration of the 4-way SMT Machine

Classification	Mix	Benchmarks
2 Low IPC	Mix 2-1	perlbmk, eon
2 Mid IPC	Mix 2-2	fma3d, sixtrack
2 High IPC	Mix 2-3	bzip2, mcf
1 Low IPC + 1 Mid IPC	Mix 2-4	gcc, sixtrack
1 Low IPC + 1 High IPC	Mix 2-5	eon, wupwise
1 Mid IPC + 1 High IPC	Mix 2-6	gzip, twolf

Table 4.2: 2-Threaded Workloads

Classification	Mix	Benchmarks
3 Low IPC	Mix 3-1	ammp, eon, vpr
3 Mid IPC	Mix 3-2	equake, fma3d, gzip
3 High IPC	Mix 3-3	lucas, mesa, mcf
1 Low IPC + 1 Mid IPC + 1 High IPC	Mix 3-4	parser, gzip, bzip2
1 Low IPC + 2 Mid IPC	Mix 3-5	mgrid, fma3d, art
2 Mid IPC + 1 High IPC	Mix 3-6	apsi, vortex, twolf
2 Low IPC + 1 High IPC	Mix 3-7	ammp, gcc, mcf
1 Low IPC + 2 High IPC	Mix 3-8	vpr, bzip2, swim
2 Low IPC + 1 Mid IPC	Mix 3-9	ammp, parser, fma3d
1 Mid IPC + 2 High IPC	Mix 3-10	apsi, swim, lucas

Table 4.3: 3-Threaded Workloads

Classification	Mix	Benchmarks
4 Low IPC	Mix 4-1	perlbmk, mgrid, ammp, parser
4 Mid IPC	Mix 4-2	sixtrack, art, apsi, applu
2 Low IPC + 2 Mid IPC	Mix 4-3	gcc, mgrid, vortex, sixtrack
4 High IPC	Mix 4-4	wupwise, galgel, bzip2, mcf
2 Mid IPC + 2 High IPC	Mix 4-5	sixtrack, fma3d, lucas, mesa
1 Low IPC + 1 Mid IPC + 2 High IPC	Mix 4-6	vpr, applu, galgel, wupwise
2 Low IPC + 2 High IPC	Mix 4-7	ammp, vpr, mcf, twolf
2 Low IPC + 1 Mid IPC + 1 High IPC	Mix 4-8	parser, perlbmk, equake, lucas
1 Low IPC + 2 Mid IPC + 1 High IPC	Mix 4-9	eon, apsi, facerec, swim

Table 4.4: 4-Threaded Workloads

Chapter 5

Evaluating the Impact of Disabling Speculative Execution on SMT

As discussed in the earlier chapters, branch prediction allows the processor to speculate the direction of a branch and speculatively execute instructions following the branch allowing the exploitation of instruction level parallelism and increasing performance. In a single-threaded superscalar processor, having a highly-accurate branch predictor and relying on aggressive speculation is one of the keys to obtaining high performance.

Figure 5.1 shows the impact of not using speculative execution (e.g., stalling and waiting till the branch is resolved) on the performance of a superscalar processor. On the average across the simulated benchmarks without prediction, the performance loss is 64.58%, ranging from 2.05% for the applu benchmark to 84.56% for the bzip2 benchmark.

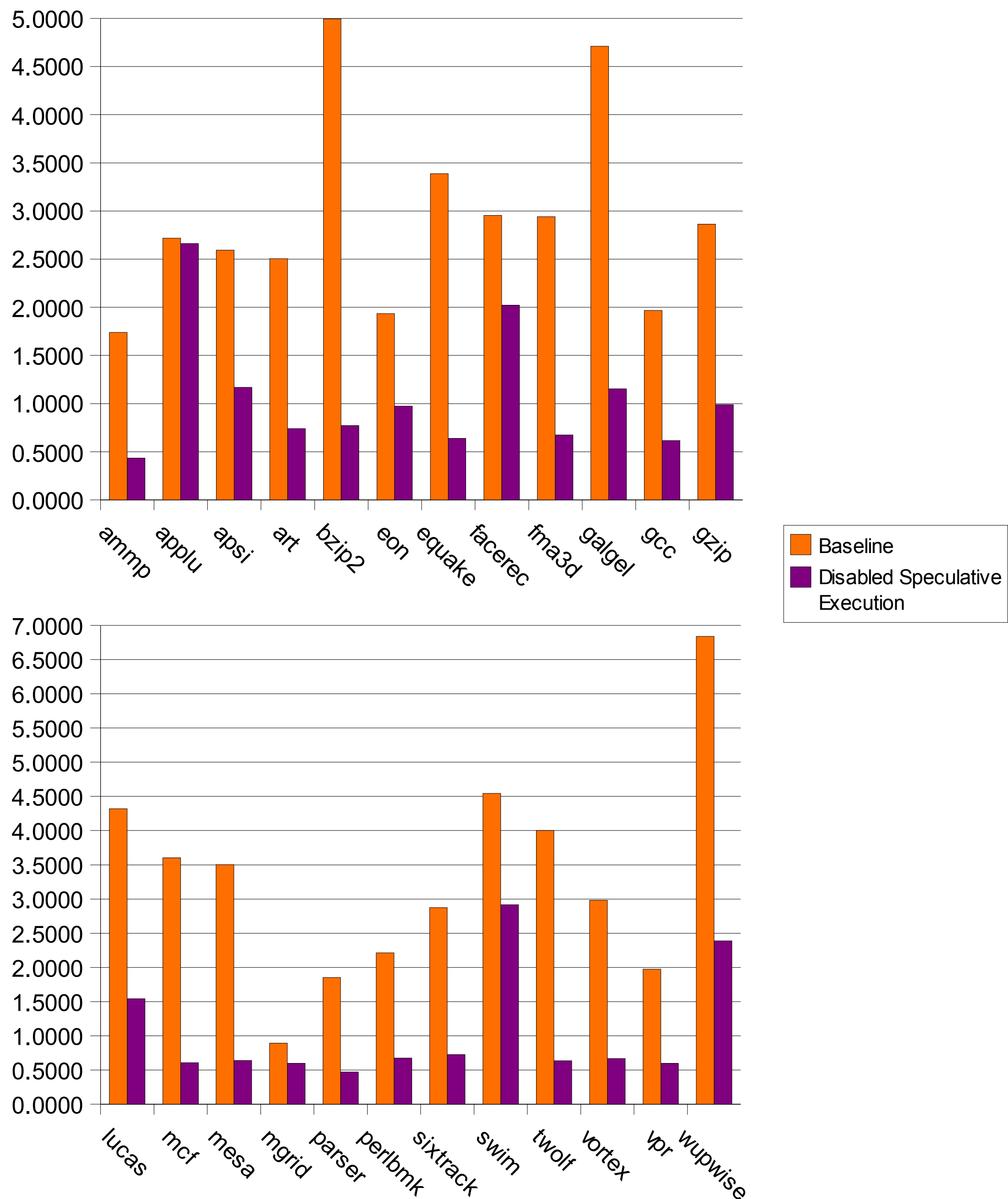


Figure 5.1: Impact of Disabling Speculative Execution for Superscalar Processors

It is clear that in the single threaded case speculation provides a consistent benefit. This is because the highly-likely-to-be-useful work can be performed instead of simply stalling

the front-end and waiting for the branch to resolve. Speculation past the unresolved branches is thus necessary for extracting the ILP from single-threaded code – a well-known fact just corroborated by these experiments.

However, when multiple thread contexts are available on SMT, the available TLP can potentially reduce the lost cycles spent waiting for the branch resolutions, and therefore stalling can be considered. Specifically, if a thread encounters a conditional branch and if this thread is stalled until the branch is resolved, then potentially non-speculative instructions from other threads can be executed in the meantime, if such instructions are available. In theory, aside from reducing the design complexity and eliminating the power consumption needed to support branch predictions, execution of the wrong-path instructions, and the activities needed to flush the pipeline and restore the precise state on mispredictions, this mechanism could also increase the performance of an SMT processor if the supply of threads is plentiful, as the processor resources will always be allocated to the correct-path instructions.

5.1 DIsabling SPeculative EXecution (DISPEX): The Implementation

When a thread encounters a conditional branch instruction (determined by using the pre-decoded opcode bits available within the fetch stage), further fetches from this thread stop by setting the *Fetch_Disable* bit (one such bit is needed for each thread). When this branch instruction is resolved, further fetches from the thread continue (the *Fetch_Disable* bit is reset) from the instructions along the correct path. As only one unresolved branch instruction can be in-flight from each thread at the same time,

determining when to re-enabling fetching is easy. When the branch instruction completes execution, its thread id (available from the issue queue entry) along with the computed address of the next instruction are sent to the fetch logic. The corresponding F*etch_Disable* bit is reset and the fetching resumes from the computed address. This eliminates the need for branch prediction and branch misprediction handling, as only the correct-path instructions are executed.

It is also possible to reduce the number of cycles for which the thread is stalled by using the dedicated logic to compute the branch outcome during the dispatch stage. After the branch undergoes register renaming, the source physical registers can be compared immediately (if they are ready), without inserting the branch into the issue queue and possibly experiencing long delays in the issue queue. In this fashion, the *Fetch_Disable* bits can be reset potentially much earlier. A same cycle computation of the branch outcome is used for the results reviewed within this chapter. Of course, if the source physical registers are not ready at the time of dispatch, then the branch has to be inserted into the issue queue and must be processed in the regular manner by the dynamic scheduling engine. This DEdicated Branch Resolution Logic (DEBRL) is used with DISPEX and DISPEX+SF.

5.2 DISPEX with Speculative Fetch (DISPEX+SF)

Simply disabling speculative execution demonstrates the need for branch prediction in order to improve instruction level parallelism. While a thread is waiting for a branch to resolve it can speculatively fetch, but not execute, instructions using a branch predictor

(or a default prediction such as *always_taken* or *never_taken*). These instructions remain in the instruction fetch queue (IFQ) until it can be determined if they are correct path instructions or need to be flushed from the IFQ. If the prediction is correct the thread can resume execution and avoid fetch-to-rename delays for the instructions that were speculatively fetched. When incorrect the IFQ will need to be flushed to maintain precise state; this penalty is less significant than branch misprediction handling since none of these instructions had been issued. The same high prediction accuracies that make branch prediction desirable on simplescalar apply to this method as well. It is expected that the IFQ will usually be filled with correct path instructions providing ample opportunity for instruction level parallelism.

When a thread encounters and renames a conditional branch instruction (determined by using dedicated logic to decode each instruction's opcode bits during register rename), further dispatches (after the branch) from this thread stop by setting the *Dispatch_Disable* bit (one such bit is needed for each thread) causing instructions to fill the IFQ. When this branch instruction is resolved, further dispatches from the thread continue (the *Dispatch_Disable* bit is reset) from the instructions along the correct path. If the original prediction was incorrect (the predicted direction can be stored using one bit for each thread) then the IFQ must be flushed of wrong-path instructions before further instructions can be fetched otherwise instructions can be dispatched directly from the IFQ. This technique only permits one unresolved branch instruction to be in the issue queue from each thread at the same time; this makes it easy to determine when to re-enable dispatching. When the branch instruction completes execution, its thread id

(available from its issue queue entry) along with the computed address of the next instruction are compared to the stored prediction. If correct, the corresponding *Dispatch_Disable* bit is reset and the dispatch and fetch stages continue normally. Otherwise, the IFQ must incur a penalty to flush wrong-path instructions and update the program counter using the computed address.

Unlike DISPEX alone, this solution uses a branch predictor to utilize the IFQ with the expectation of a successful prediction often enough to justify the penalty due to flushing the front-end and undoing incorrect register renaming (two cycles) when incorrect instructions are speculatively fetched. Traditional branch misprediction recovery schemes are not needed with this solution since only correct-path instructions are executed.

5.3 DISPEX+SF with MICOUNT and ADEBRL (Aggressive DISPEX+SF)

Seeking further performance improvements DISPEX+SF will be made aggressive (ADISPEX+SF) by two additional modifications; aggressive DEBRL (ADERBL) and modified ICOUNT (MICOUNT).

ADEBRL is added to DISPEX+SF to quickly resolve a branch that generates a stall, thus reducing the requirement to wait until the branch executes. DEBRL responds to a stall by either resolving the branch or waiting it to execute and doing nothing. The aggressive implementation, ADEBRL, continues to attempt to resolve the branch while waiting for it to execute. In all cases this should improve performance since it can not extend the stall length and only reduce it.

When instructions are fetched from threads that are currently stalled a performance loss occurs. These fetched instructions are unable to execute and waste fetch bandwidth that could fetch instructions from other threads on an SMT machine. MICOUNT is a modification of ICOUNT that disregards any thread with a *Fetch_Disable* or *Dispatch_Disable* bit set. MICOUNT must select two threads, although they can be the same, if all threads have a *Fetch_Disable* or *Dispatch_Disable* bit set then MICOUNT selects two threads as if it were not modified.

5.4 Results of DISPEX, DISPEX+SF and ADISPEX+SF

Figures 5.2-5.4 present the performance of DISPEX (as described in section 5.1). The results are presented in terms of throughput IPC. For 2-threaded workloads (Figure 5.2) there is an average performance loss of 61.31% ranging from 37.5% for Mix 2-5 to 75.27% for Mix 2-3. For 3-threaded workloads (Figure 5.3) there is an average performance loss of 57.48% ranging from 30.13% for Mix 3-10 to 68.21% for Mix 3-4. For 4-threaded workloads (Figure 5.4) there is an average performance loss of 46.7% ranging from 23.44% for Mix 4-9 to 63.74% for Mix 4-7.

Figures 5.2-5.4 also present the performance of DISPEX+SF (as described in section 5.2). The results are presented in terms of throughput IPC. For 2-threaded workloads (Figure 5.2) there is an average performance loss of 45.38% ranging from 15.74% for Mix 2-5 to 63.14% for Mix 2-3. For 3-threaded workloads (Figure 5.3) there is an average performance loss of 35.41% ranging from 8.78% for Mix 3-10 to 51.66% for Mix 3-4.

For 4-threaded workloads (Figure 5.4) there is an average performance loss of 28.57% ranging from 12.76% for Mix 4-9 to 48.27% for Mix 4-9.

Figures 5.2-5.4 also present the performance ADISPEX+SF (as described in section 5.3). The results are presented in terms of throughput IPC. For 2-threaded workloads (Figure 5.2) there is an average performance loss of 43.13% ranging from 14.39% for Mix 2-5 to 62.78% for Mix 2-3. For 3-threaded workloads (Figure 5.3) there is an average performance loss of 33.91% ranging from 5.85% for Mix 3-10 to 50.8% for Mix 3-4. For 4-threaded workloads (Figure 5.4) there is an average performance loss of 24.61% ranging from 5.36% for Mix 4-9 to 49.32% for Mix 4-7.

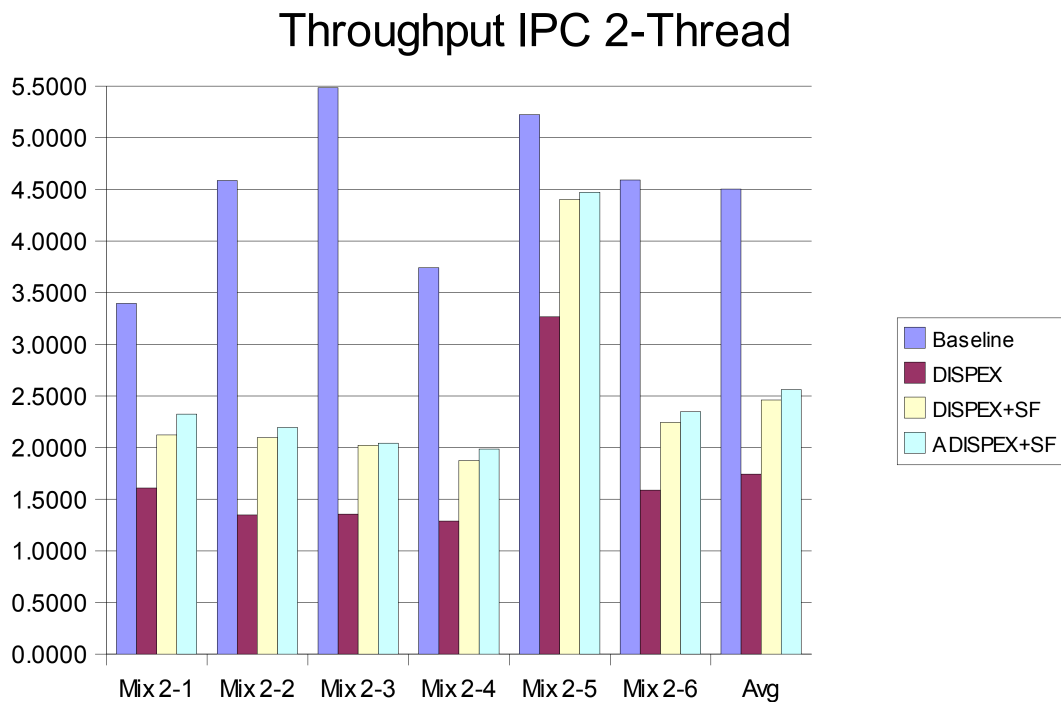


Figure 5.2: Throughput IPC for 2-threaded Workloads with DISPEX, DISPEX+SF and ADISPEX+SF

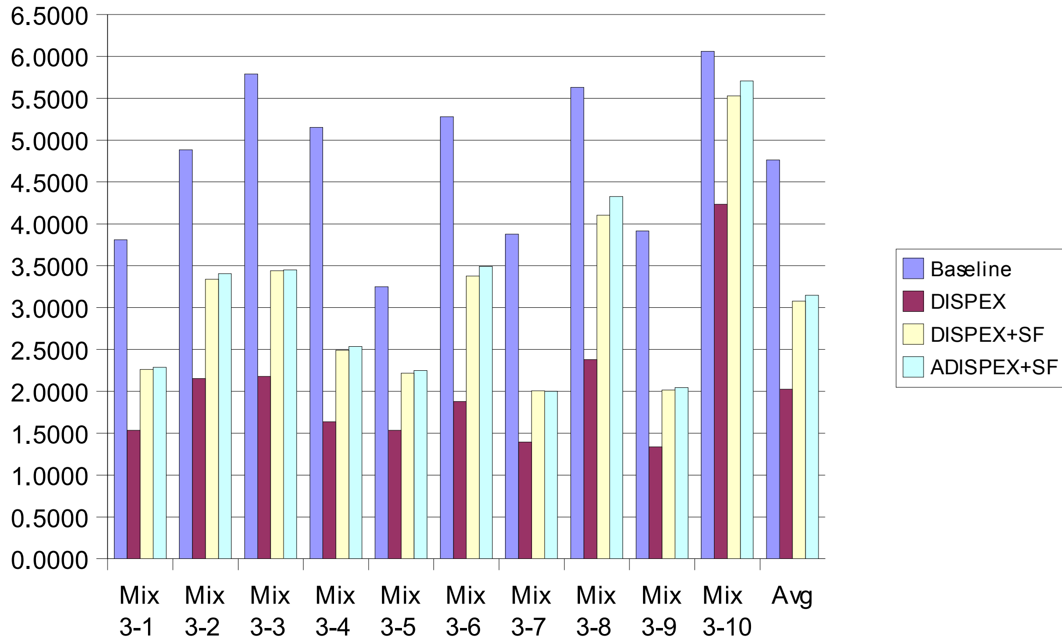


Figure 5.3: Throughput IPC for 3-threaded Workloads with DISPEX, DISPEX+SF and ADISPEX+SF

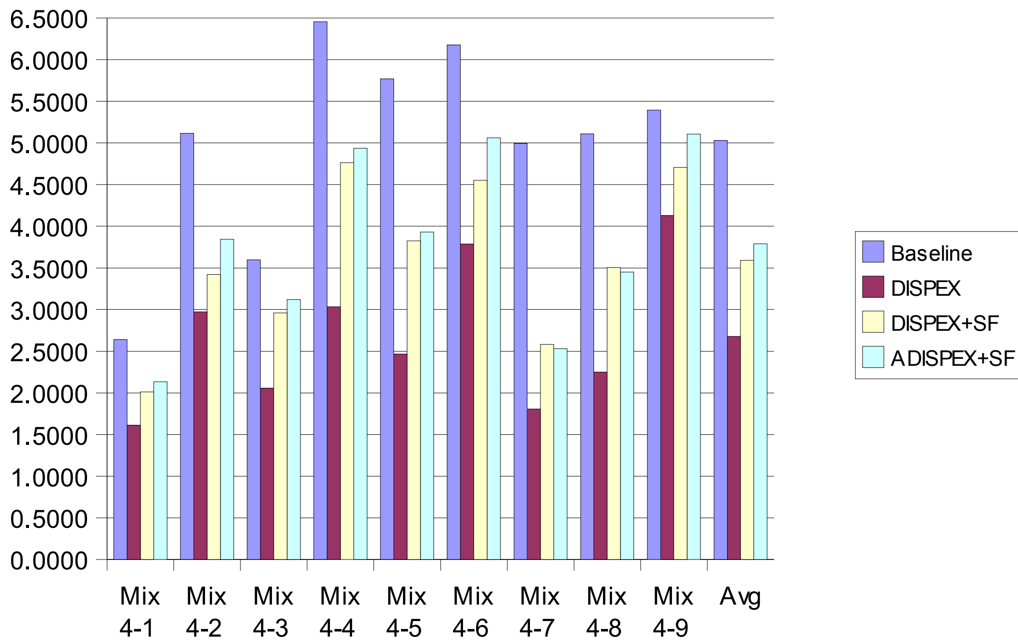


Figure 5.4: Throughput IPC for 4-threaded Workloads with DISPEX, DISPEX+SF and ADISPEX+SF

Figures 5.5-5.7 present the performance of DISPEX (as described in section 5.1). The results are presented in terms of fairness. For 2-threaded workloads (Figure 5.5) there is an average fairness loss of 62.23% ranging from 38.05% for Mix 2-5 to 75.3% for Mix 2-3. For 3-threaded workloads (Figure 5.6) there is an average fairness loss of 59.37% ranging from 31.22% for Mix 3-10 to 68.29% for Mix 3-4. For 4-threaded workloads (Figure 5.7) there is an average fairness loss of 46.02% ranging from 23.51% for Mix 4-9 to 64.15% for Mix 4-7.

Figures 5.5-5.7 also present the performance of DISPEX+SF (as described in section 5.2). The results are presented in terms of fairness. For 2-threaded workloads (Figure 5.5) there is an average fairness loss of 46.55% ranging from 17.19% for Mix 2-5 to 63.16% for Mix 2-3. For 3-threaded workloads (Figure 5.6) there is an average fairness loss of 40.89% ranging from 10.33% for Mix 3-10 to 51.59% for Mix 3-4. For 4-threaded workloads (Figure 5.7) there is an average fairness loss of 29.61% ranging from 13.61% for Mix 4-9 to 48.94% for Mix 4-7.

Figures 5.5-5.7 also present the performance of ADISPEX+SF (as described in section 5.3). The results are presented in terms of fairness. For 2-threaded workloads (Figure 5.5) there is an average fairness loss of 44% ranging from 15.16% for Mix 2-5 to 62.79% for Mix 2-3. For 3-threaded workloads (Figure 5.6) there is an average fairness loss of 40.48% ranging from 8.38% for Mix 3-10 to 50.78% for Mix 3-4. For 4-threaded workloads (Figure 5.7) there is an average fairness loss of 25.93% ranging from 5.96% for Mix 4-9 to 50.33% for Mix 4-7.

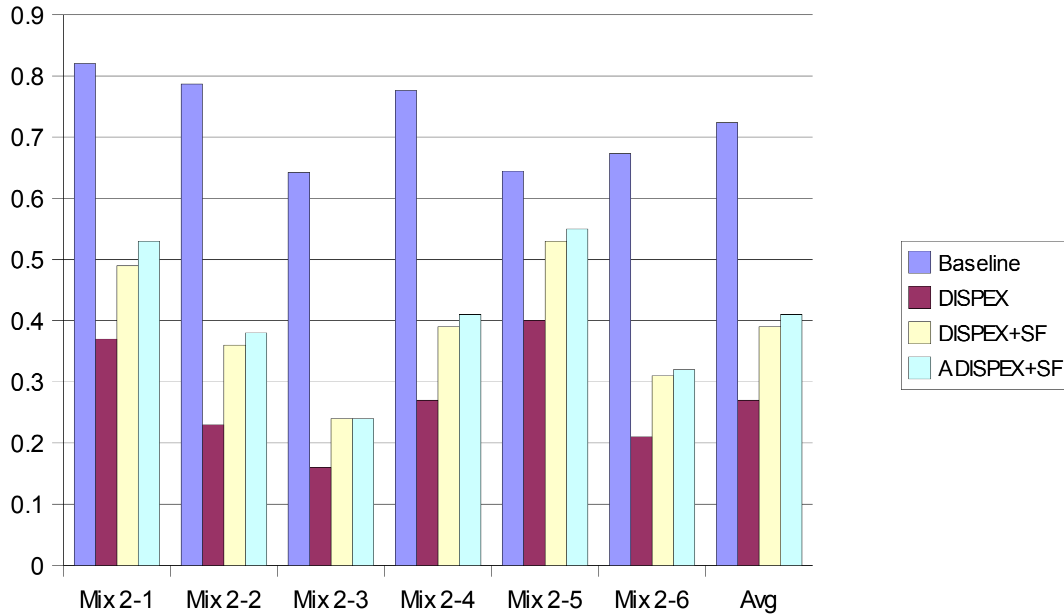


Figure 5.5: Fairness for 2-threaded Workloads with DISPEX, DISPEX+SF and ADISPEX+SF

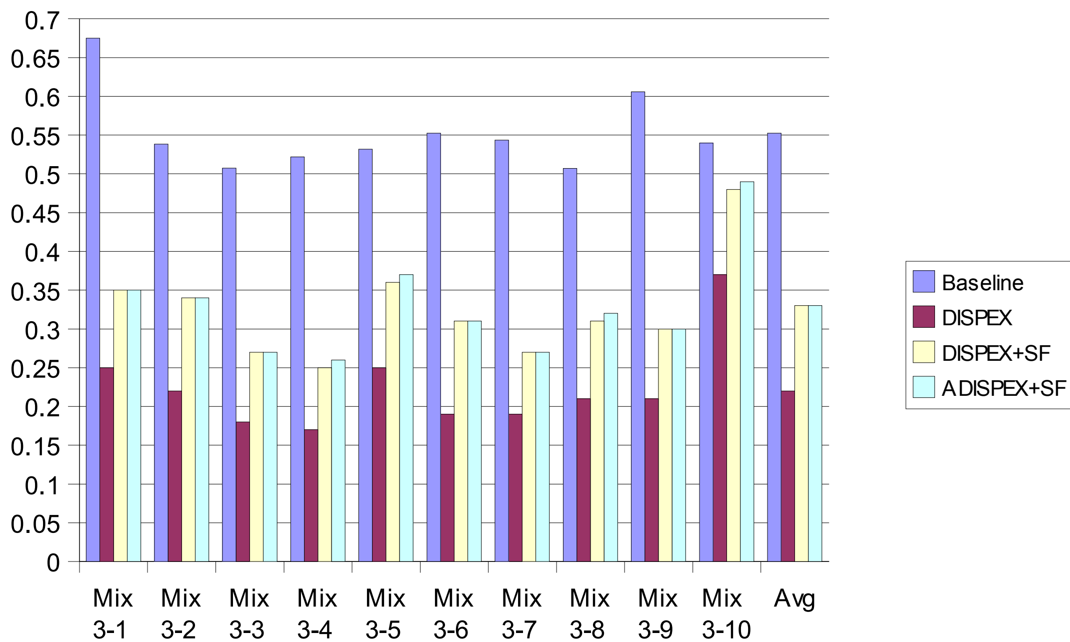


Figure 5.6: Fairness for 3-threaded Workloads with DISPEX, DISPEX+SF and ADISPEX+SF

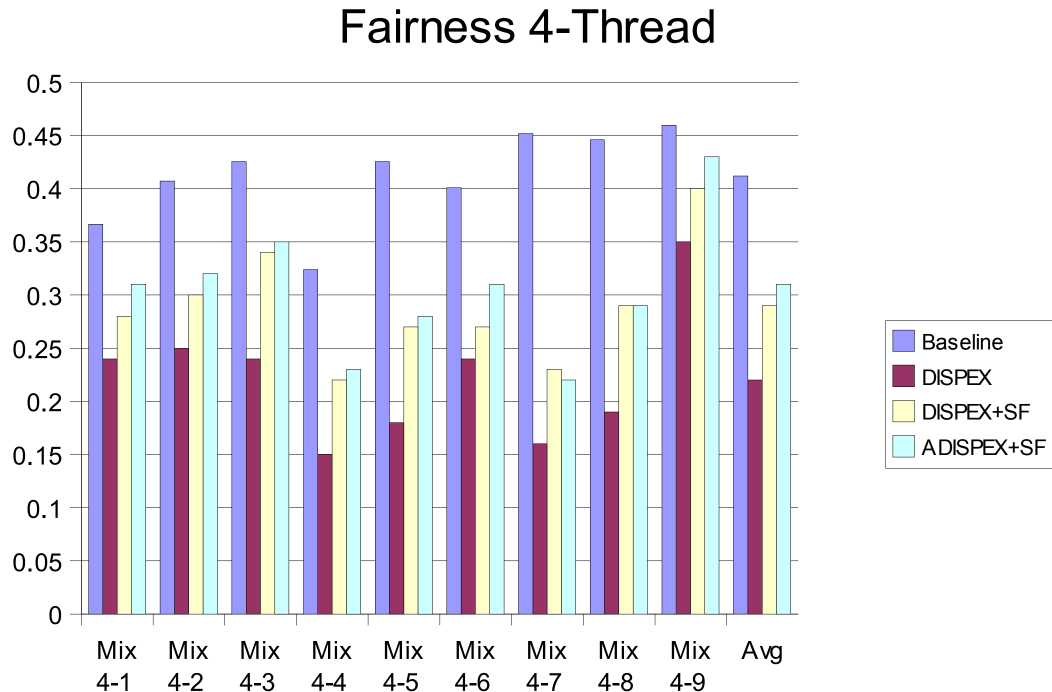


Figure 5.7: Fairness for 4-threaded Workloads with DISPEX, DISPEX+SF and ADISPEX+SF

The results of DISPEX demonstrate that speculative execution significantly improves thread-level parallelism among the workloads. However, the TLP derived from the execution of multiple threads is not enough to hide the latency generated from stalling until branch direction resolution. DISPEX+SF still suffers from an inability to hide branch resolution latency but performs significantly better than stalling the fetcher in terms of both throughput IPC and fairness. Comparing DISPEX and DISPEX+SF clearly shows that DISPEX+SF, which requires a branch predictor but not traditional misprediction handling, performs better than DISPEX in all cases; Throughput IPC losses for two threads 61.31%/45.38% for three threads 54.78%/35.41% and for four threads 46.70%/26.57%.

As expected, making DISPEX+SF more aggressive (ADISPEX+SF) showed performance

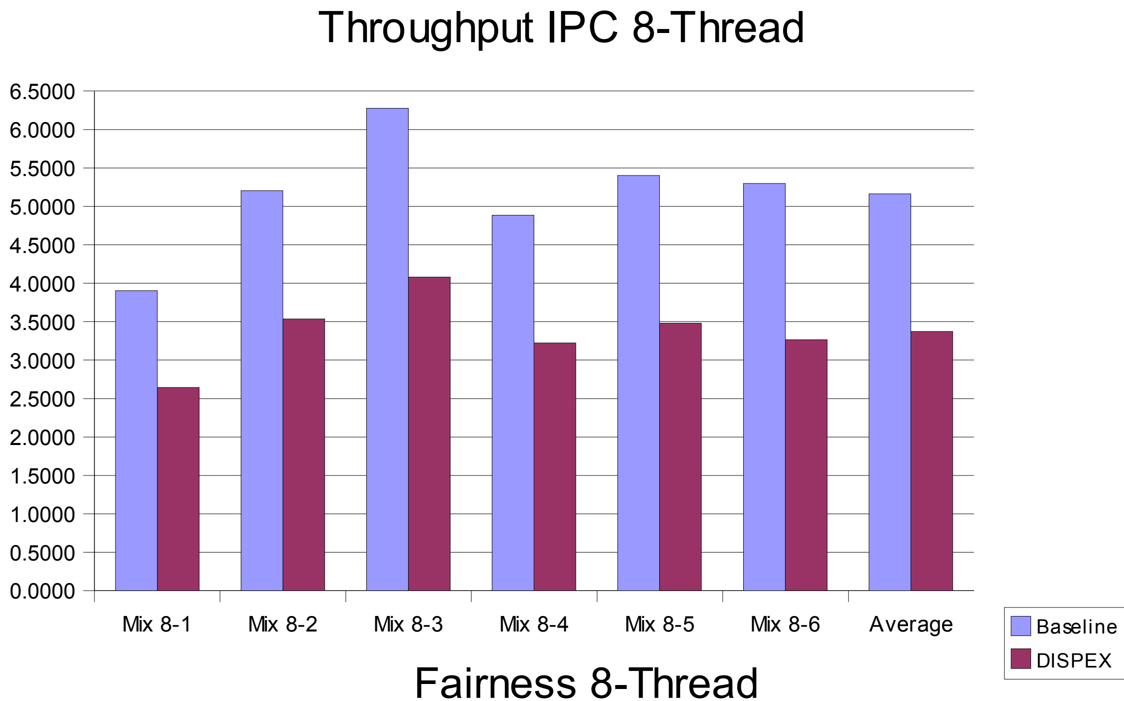
gains in all cases; Throughput IPC gains for ADISPEX+SF compared to DISPEX+SF were 4.11% for two threads, 2.33% for three threads and 5.54% for four threads.

These results demonstrate that thread level parallelism can hide some latency, but not all of it. DISPEX shows that adding support for an additional thread context reduces the performance loss compared to the baseline scenario; 61.31% for two threads, 54.78% for three threads, 46.70% for four threads, and 34.55% for eight threads. Similar results can be shown for DISPEX+SF; 45.38% for two threads to 35.41% for three threads and 28.57% for four threads.

The inability of thread level parallelism to cover up branch resolution latency may come from various sources. The fetching policy, ICOUNT, is not adequate for use with these techniques. A stalled thread may be allocated fetch resources but be unable to use them resulting in a loss of use of these resources by other threads. This problem is due to how ICOUNT works; by selecting from threads with less in-flight and not-yet-issued instructions, stalling threads reduces the number of in-flight instructions for threads that can not be progressed. The DISPEX+SF is slightly less prone to this problem because speculative fetching into the IFQ is expected to improve performance for the thread and can reduce the possibility of the thread being selected while it is stalled. The problem still exists and performance may be further improved by modifying ICOUNT to better accommodate the stalling of threads.

In order to demonstrate that additional thread context support improves thread-level

parallelism DISPEX was applied to 8-threaded workloads using the same implementation; except allowing 8 threads. The results for 8-threaded workloads are shown in Figure 5.8 (with workload contents in Table 5.1) with an average throughput IPC loss of 34.55% ranging from 32.06% for Mix 8-2 to 38.39% for Mix 8-6. Fairness losses average performance loss of 36.05% ranging from 28.82% for Mix 8-1 to 43.18% for Mix 8-6. These results clearly demonstrate that even eight threads cannot hide this latency and even an aggressive design may not be able to overcome this performance penalty.



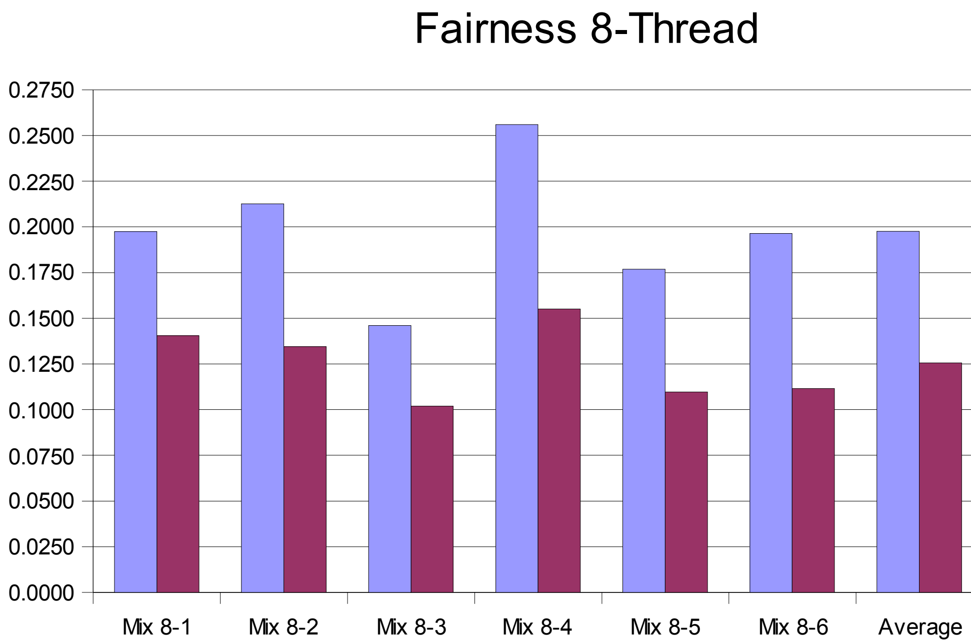


Figure 5.8: Performance of 8-threaded Workloads with DISPEX

Exploitable TLP from an SMT machine is not enough to cover up the huge performance losses that are generated by DISPEX. Performance improves as thread context support increases but even with 8-thread support losses were as high as 38.39%. Speculative fetch reduced the performance loss but losses still remained excessive. The next section

describes modifications to DISPEX that attempt to reduce the performance losses described in this section.

Classification	Mix	Benchmarks
7 Low IPC + 1 High IPC	Mix 8-1	mgrid, ammp, parser, eon, gcc, vpr, perlbmk, art
8 Mid IPC	Mix 8-2	apsi, applu, gzip, sixtrack, fma3d, facerec, vortex, equake
8 High IPC	Mix 8-3	mesa, mcf, twolf, lucas, swim, galgel, bzip2, wupwise
3 Low IPC + 4 Mid IPC + 1 High IPC	Mix 8-4	gcc, vpr, perlbmk, art, apsi, applu, gzip, sixtrack
4 Mid IPC + 4 High IPC	Mix 8-5	fma3d, facerec, vortex, equake, mesa, mcf, twolf, lucas
3 Low IPC + 1 Mid IPC + 4 High IPC	Mix 8-6	gcc, vpr, perlbmk, art, mesa, mcf, twolf, lucas

Table 5.1: 8-Threaded Workloads

5.5 DYnamically COntrolled SPeculative EXecution (DYCOSPEX)

DISPEX and DISPEX+SF are hindered by the common occurrence of all threads being stalled waiting for the resolution of a branch instruction, even the more aggressive ADISPEX+SF does not perform well. This is demonstrated by observing the fetch behavior when DISPEX is implemented. For 2-threaded workloads 62.35% of all fetches are lost because all threads are stalled. As the number of thread contexts increases, TLP is able to hide these losses but they are still significant: 55.88% for 3-threaded workloads and 46.73% for 4-threaded workloads. In these cases the ability to bypass a stall, in order to avoid excessive performance loss, is permitted when all threads are stalled.

DYCOSPEX is the combination of DISPEX with MICOUNT and ADEBRL with the ability to dynamically control the activation and deactivation of speculative execution bypassing branch stalls. This is done to prevent situations where all threads stall and no work is completed. When a thread stalls on some conditional branch instruction it can be bypassed as long as it has not yet been renamed; otherwise more than one unresolved

branch instruction can be in-flight at the same time complicating mechanisms that restore the precise state in case of a misprediction as well as making it more difficult to determine when to re-enable fetching. Bypassing occurs when all threads are stalled (this is detected by checking the *Fetch_Disable* bits of the threads selected by MICOUNT) and the threads selected by MICOUNT are eligible for bypassing (this is accomplished with a single bit for each thread, *Cannot_Bypass* which is set when a conditional branch instruction is renamed and cleared when *Fetch_Disable* is set for the same thread). Bypass causes the *Fetch_Disable* bit for the appropriate thread to be cleared and fetching continues from the predicted program counter. This requires traditional branch misprediction hardware.

5.6 Results of DYCOPLEX

Using DYCOPLEX demonstrated that thread-level parallelism is not effective enough to cover up latency generated by stalling for branch resolution even with the inclusion of MICOUNT, ADEBRL and bypassing logic. Bypassing stalls generates counter-intuitive results; appearing to degrade when thread-level parallelism is first increased. This occurs due to 2-threaded workloads behaving similarly to the baseline case. When there are only two threads, only three cases exist; both threads active, one thread stalled, both threads stalled. When both threads are active or stalled, behavior continues as normal (both being stalled results in both bypassing their stall, there is a performance loss because speculation was not attempted immediately). When only one thread is stalled the other receives all of the fetch bandwidth but often can not use all of it due to the frequency of branch instructions (and not allowing the other thread to fetch). Mix 2-5 is an outlier,

showing a performance gain respective to the baseline case, with performance gains likely due to the inclusion of wupwise (single-threaded IPC of 6.8372) which may be able to utilize available fetch bandwidth forfeited by eon (while fairness increased (0.6444 -> 0.6884) the per-thread IPC of eon dropped (1.5636 -> 1.1461) while wupwise rose significantly (3.6609 -> 5.2361)).

Adding support for a third thread generates disappointing results as performance and fairness decrease significantly. With a third thread a different situation arises; when all three threads stall and two are selected using MICOUNT. Two of the stalled threads are bypassed by DYCOSPEX causing the third thread to become more likely to be selected. Statistically, the two bypassed threads are likely to stall again due to the frequency of branch instructions; therefore the third thread and one of the bypassed threads are both bypassed in a later cycle acting as a strict pipeline bubble and recreating the same situation. This creates a pipeline bubble for the "third thread" due to being stalled but ultimately bypassed before resolution resulting in a penalty over simply predicting and not stalling. Support for a fourth thread context improves performance but still worse than with 2-threaded workloads. Additional thread context support may provide additional TLP to reduce the throttling penalty.

Figures 5.9-5.11 present the performance of disabling speculative execution with a modified fetcher, bypass and aggressive branch resolution with same cycle computation of the branch target (as described in section 5.5). The results are presented in terms of throughput IPC. For 2-threaded workloads (Figure 5.9) there is an average performance

loss of 7.64% ranging from -22.16% for Mix 2-5 to 24.82% for Mix 2-2. For 3-threaded workloads (Figure 5.10) there is an average performance loss of 21.12% ranging from 4.96% for Mix 3-10 to 28.97% for Mix 3-7. For 4-threaded workloads (Figure 5.11) there is an average performance loss of 17.19% ranging from 1.53% for Mix 4-9 to 39.72% for Mix 4-7.

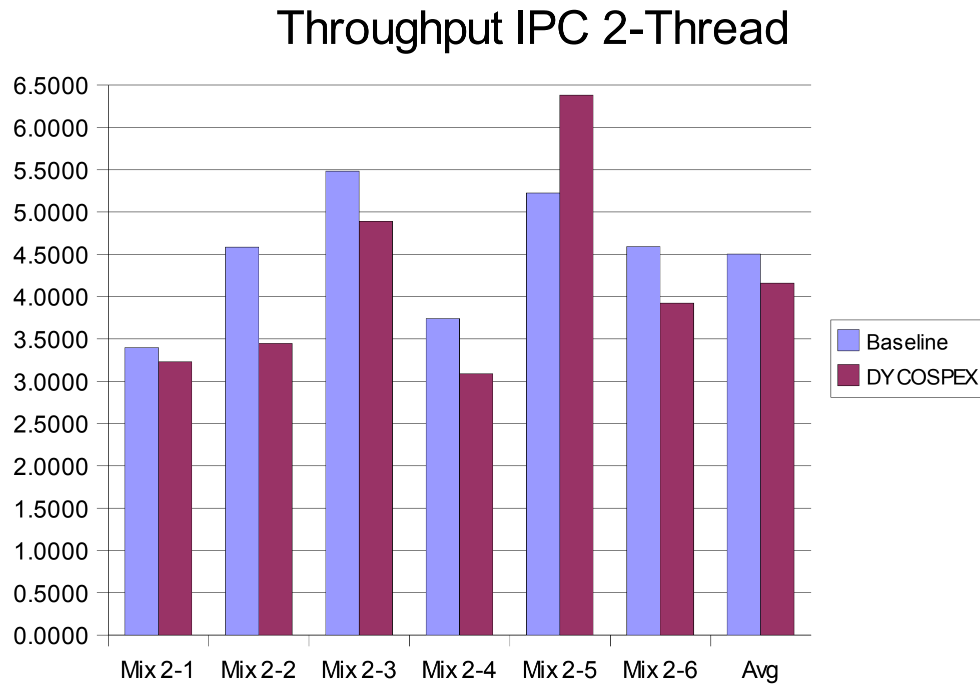


Figure 5.9: Throughput IPC for 2-threaded Workloads with DYCOSPEX

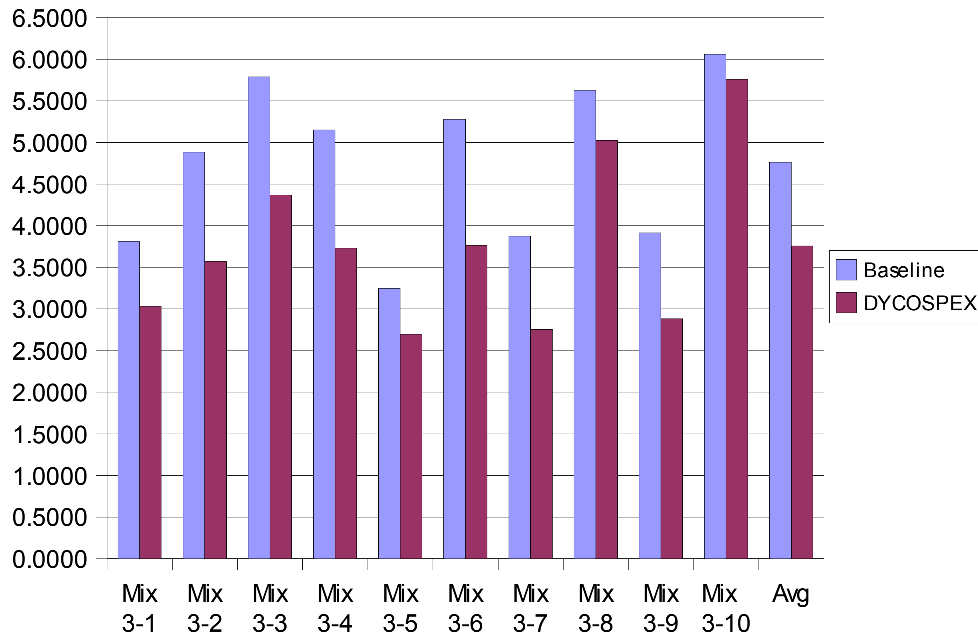


Figure 5.10: Throughput IPC for 3-threaded Workloads with DYCOSPEX

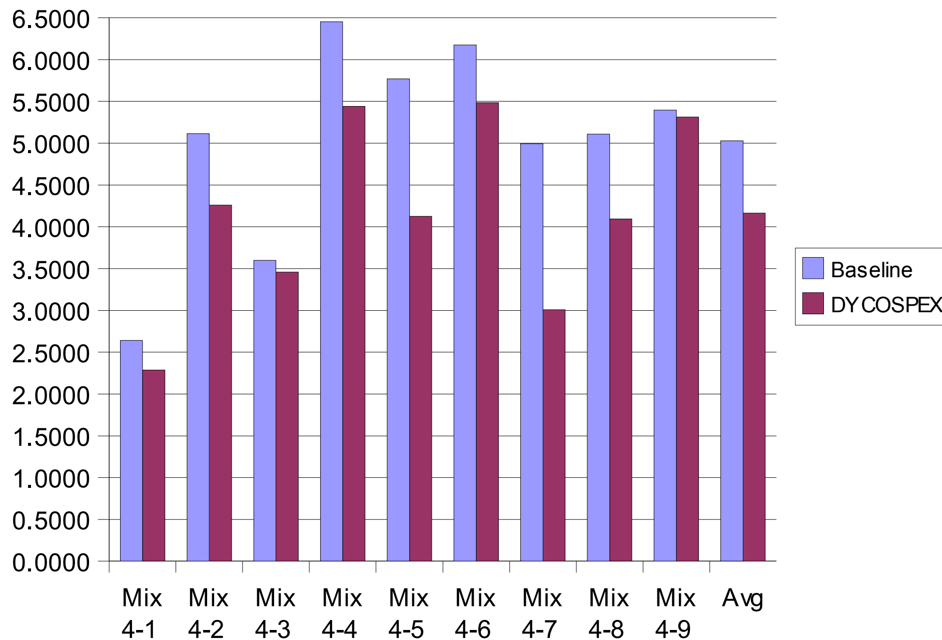


Figure 5.11: Throughput IPC for 4-threaded Workloads with DYCOSPEX

Figures 5.12-5.14 present the performance of disabling speculative execution with a

modified fetcher, bypass and aggressive branch resolution with same cycle computation of the branch target (as described in section 5.5). The results are presented in terms of fairness. For 2-threaded workloads (Figure 5.12) there is an average fairness loss of 13.02% ranging from -3.73% for Mix 2-5 to 25.25% for Mix 2-2. For 3-threaded workloads (Figure 5.13) there is an average fairness loss of 26.04% ranging from 6.57% for Mix 3-10 to 50.27% for Mix 3-2. For 4-threaded workloads (Figure 5.14) there is an average fairness loss of 23.15% ranging from 4.12% for Mix 4-9 to 39.62% for Mix 4-7.

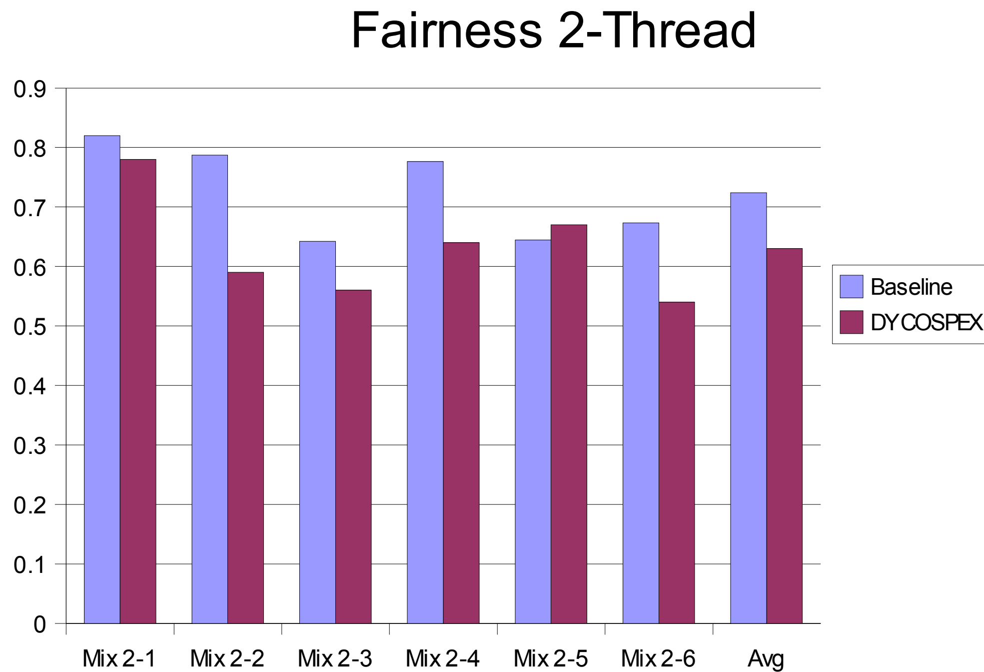


Figure 5.12: Fairness for 2-threaded Workloads with DYCOSPEX

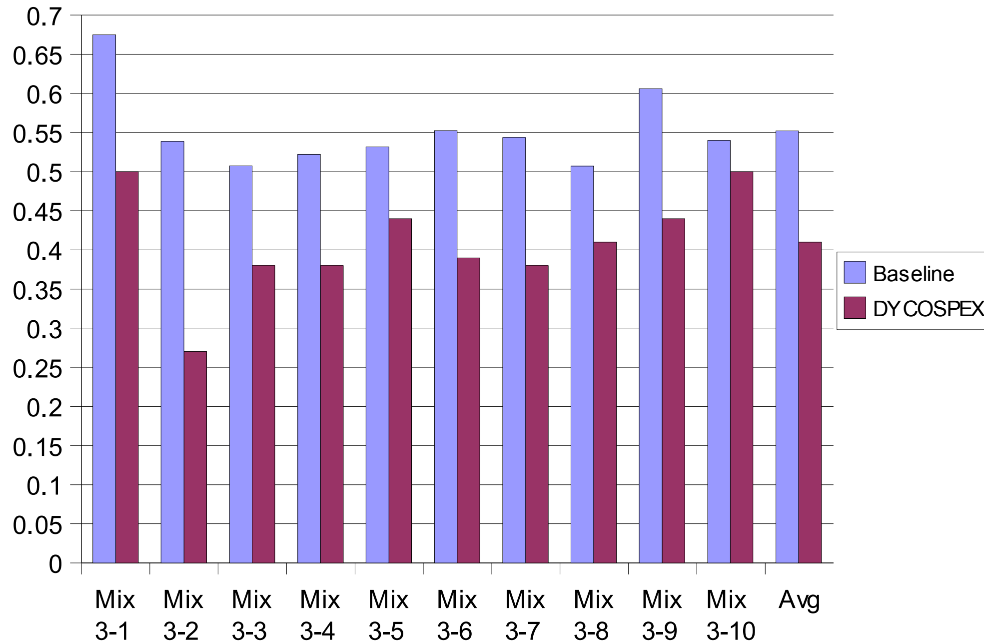


Figure 5.13: Fairness for 3-threaded Workloads with DYCOSPEX

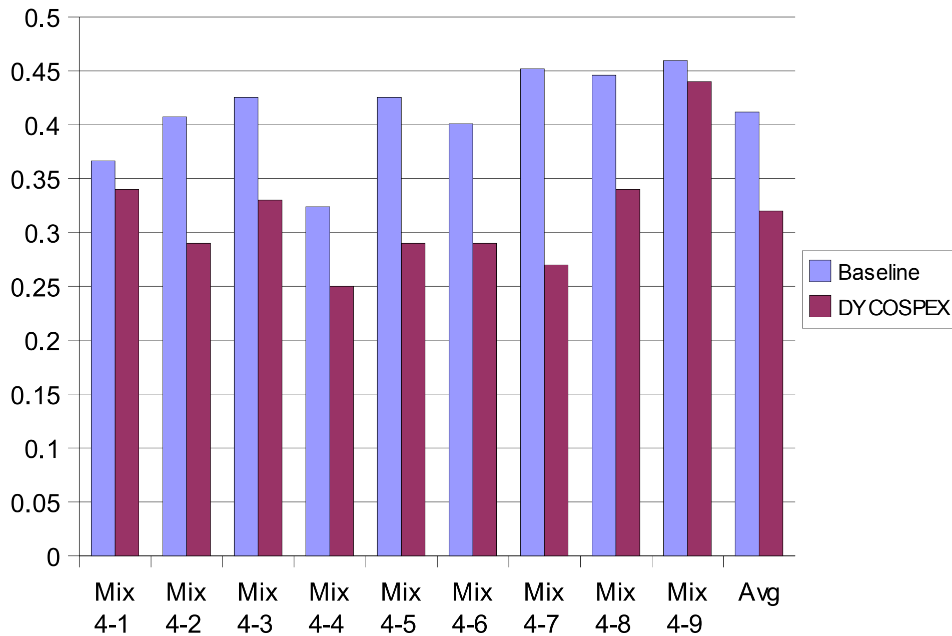


Figure 5.14: Fairness for 4-threaded Workloads with DYCOSPEX

5.7 COnfidence COntrolled SPeculative EXecution (COCOSPEX): Implementation

Prior techniques in this chapter take different approaches to determine when to disable speculative execution. The methods described in sections 5.1-5.3 disable speculative execution continuously. This contrasts with the method described in section 5.5 where speculative execution can be re-enabled to prevent complete pipeline stalling. In this section, a confidence based technique is used to determine when speculative execution is disabled (COCOSPEX).

Whenever a branch prediction is made a confidence value (*Branch_Confidence*) is returned in addition to the prediction. If the instruction fetch logic receives a *Branch_Confidence* value below the established threshold (*Confidence_Threshold*) further fetches from this thread stop by setting the *Fetch_Disable* bit (one bit is needed for each thread), otherwise the instruction proceeds normally. Once the branch instruction is resolved the *Fetch_Disable* bit can be reset for that thread (available from the issue queue entry) and further fetches can continue from the computed address.

Updates to the branch predictor tables are modified to alter the confidence value associated with the corresponding branch instruction. If the prediction was incorrect the confidence is set to 0, otherwise it is increased by 1 to a maximum value equal to *Confidence_Threshold*.

5.8 Results of COCOSPEX

Figures 5.15-5.17 present the performance of COCOSPEX (as described in section 5.7) with a *Confidence_Threshold* of 3. The results are presented in terms of throughput IPC.

For 2-threaded workloads (Figure 5.15) there is an average performance loss of 4.75% ranging from -24.30% for Mix 2-5 to 20.75% for Mix 2-2. For 3-threaded workloads (Figure 5.16) there is an average performance loss of 8.49% ranging from -0.06% for Mix 3-10 to 17.33% for Mix 3-5. For 4-threaded workloads (Figure 5.17) there is an average performance loss of 2.94% ranging from -7.74% for Mix 4-1 to 11.04% for Mix 4-7.

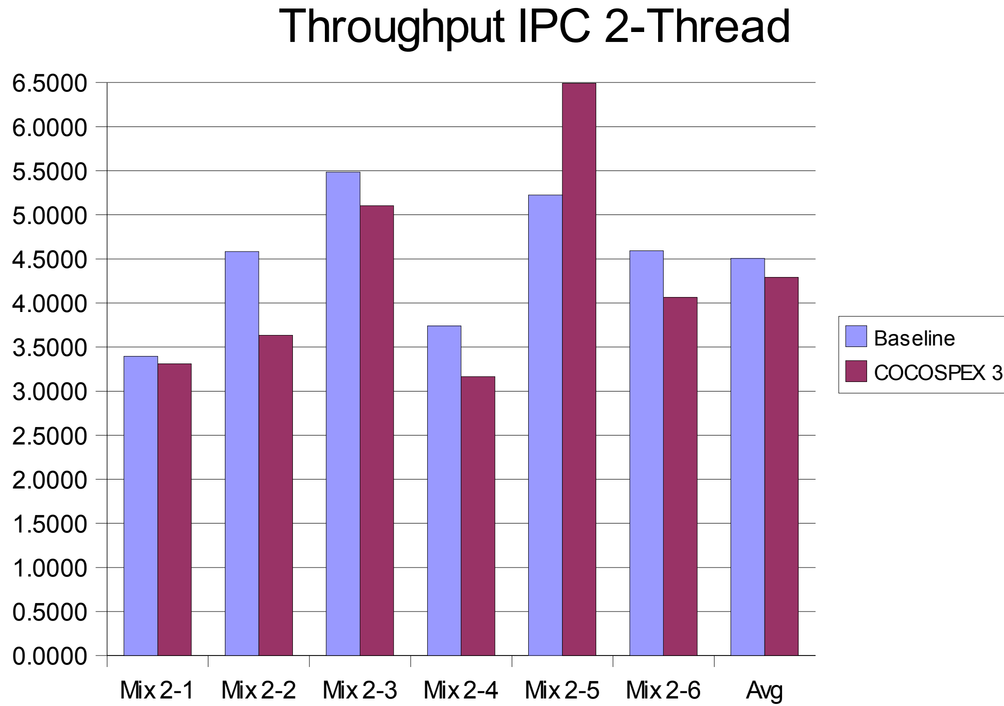


Figure 5.15: Throughput IPC for 2-threaded Workloads with COCOSPEX

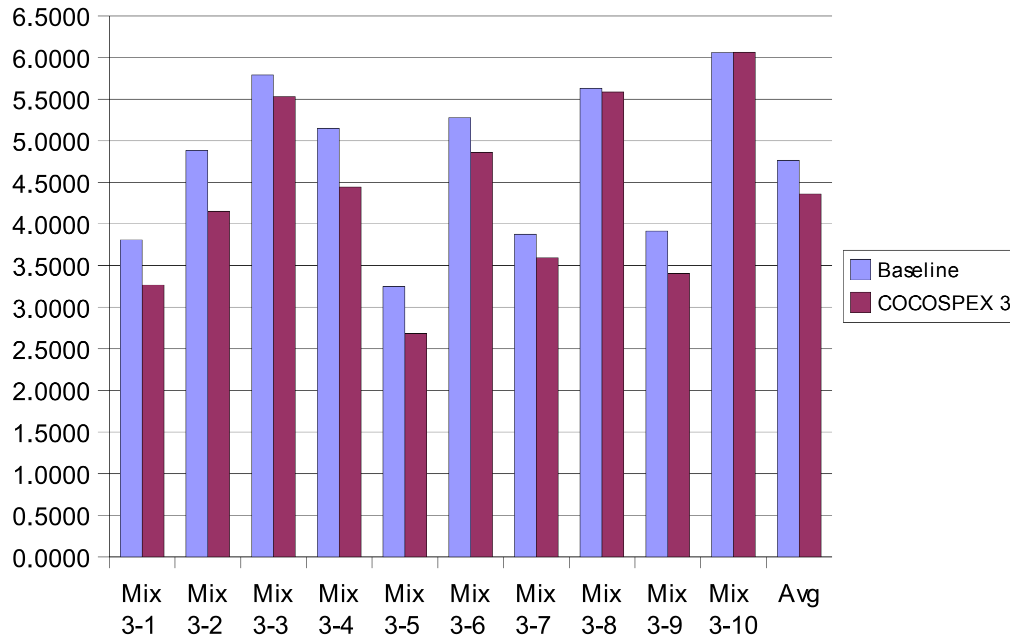


Figure 5.16: Throughput IPC for 3-threaded Workloads with COCOSPEX

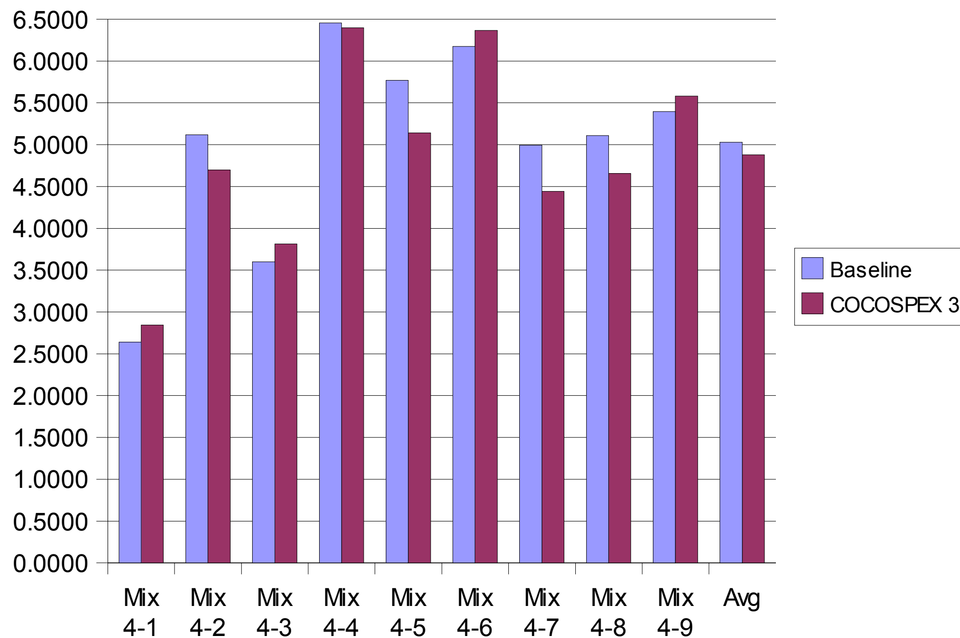


Figure 5.17: Throughput IPC for 4-threaded Workloads with COCOSPEX

Figures 5.18-5.20 present the performance of COCOSPEX (as described in section 5.7)

with a *Confidence_Threshold* of 3. The results are presented in terms of fairness. For 2-threaded workloads (Figure 5.18) there is an average fairness loss of 8.83% ranging from -10.06% for Mix 2-5 to 21.45% for Mix 2-2. For 3-threaded workloads (Figure 5.19) there is an average fairness loss of 11.28% ranging from -0.02% for Mix 3-10 to 15.34% for Mix 3-1. For 4-threaded workloads (Figure 5.20) there is an average fairness loss of 2.88% ranging from -9.74% for Mix 4-1 to 11.06% for Mix 4-5.

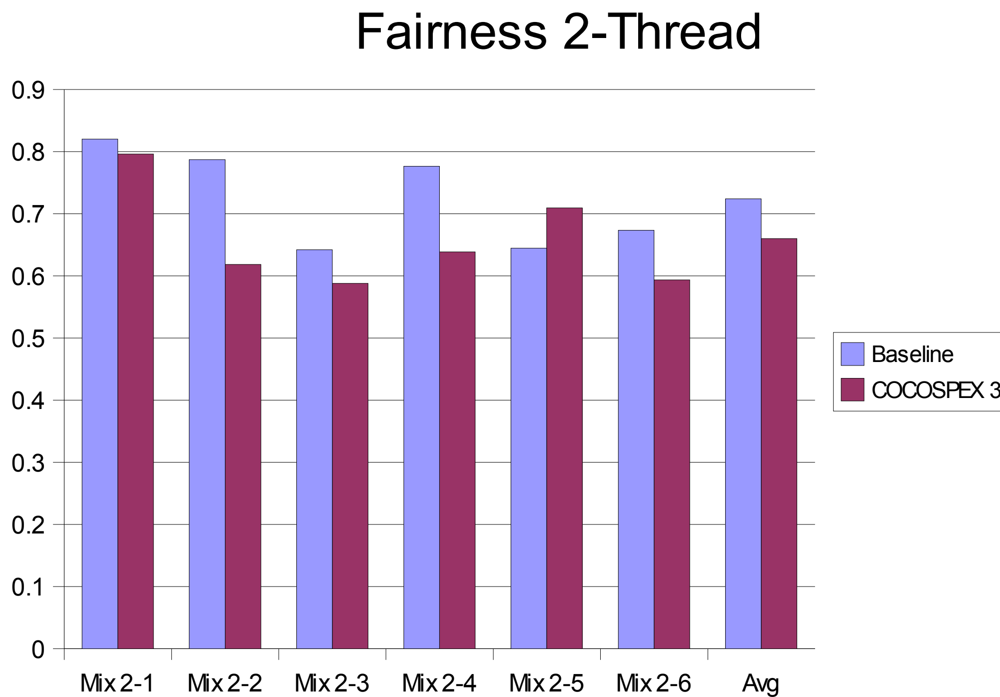


Figure 5.18: Fairness for 2-threaded Workloads with COCOSPEX

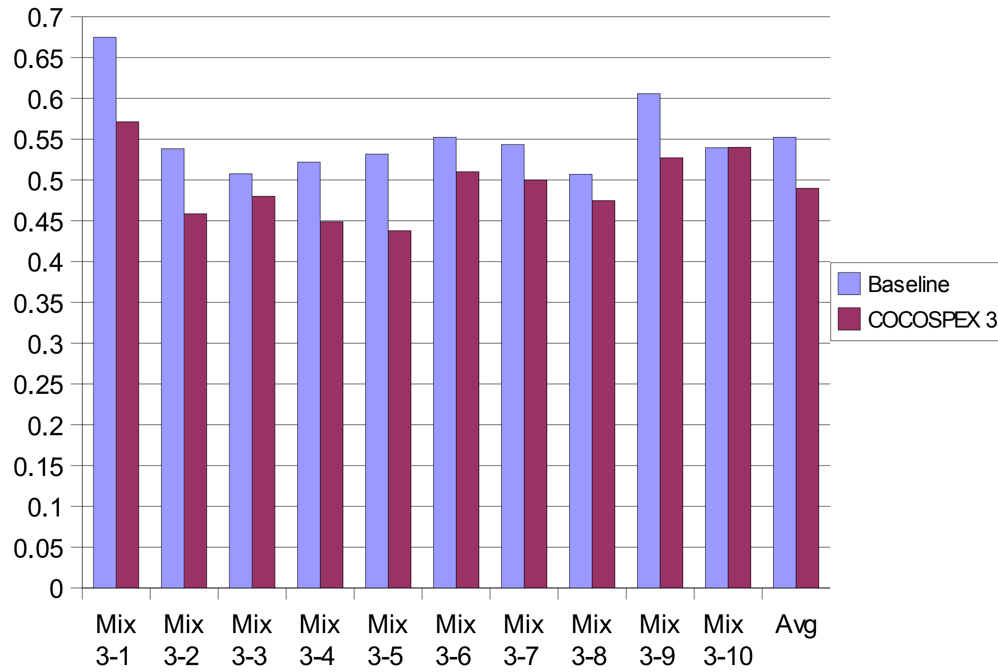


Figure 5.19: Fairness for 3-threaded Workloads with COCOSPEX

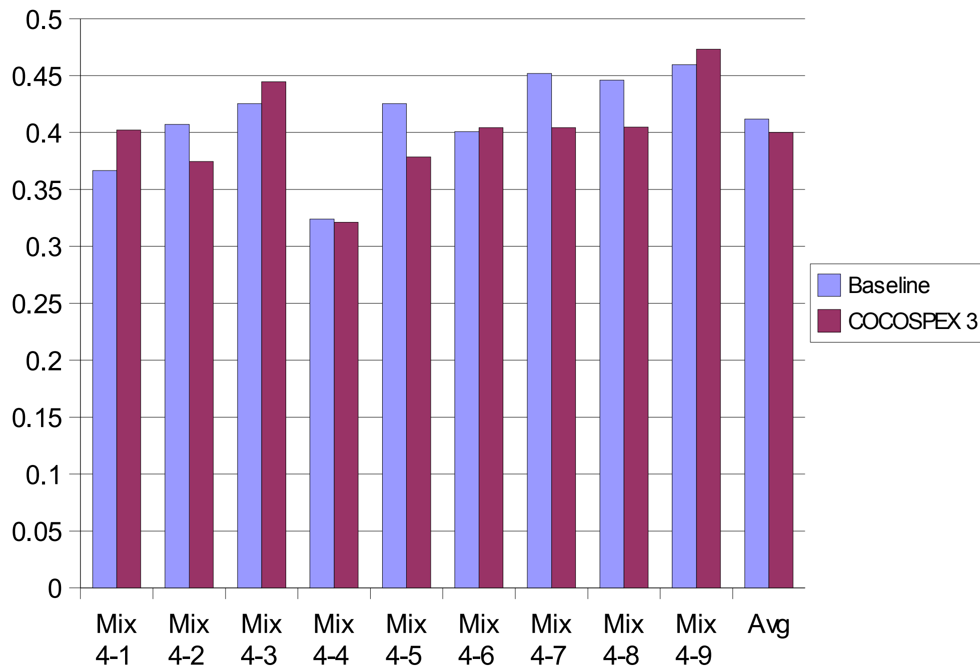


Figure 5.20: Fairness for 4-threaded Workloads with COCOSPEX

This method performs better than DISPEX and even in some mixes improves performance. Regardless, the overall performance is still worse than the baseline case. Since COCOSPEX imposes an additional hardware burden on an already tight branch predictor hardware budget and provides no overall benefit it is not suggested except for specialized use or perhaps with known program phases.

5.9 Summary

	# Threads	Average IPC Performance Loss	Min. IPC Performance Loss	Max. IPC Performance Loss	Average Fairness Loss	Min. Fairness Loss	Max. Fairness Loss
DISPEX	2	61.31%	37.50%	75.27%	62.23%	38.05%	75.30%
	3	54.78%	30.13%	68.21%	59.37%	31.22%	68.29%
	4	46.70%	23.44%	63.74%	46.02%	23.51%	64.15%
	8	34.55%	32.06%	38.39%	36.05%	28.82%	43.18%
DISPEX+SF	2	45.38%	15.74%	63.14%	46.55%	17.19%	63.16%
	3	35.41%	8.78%	51.66%	40.89%	10.33%	51.59%
	4	28.57%	12.76%	48.27%	29.61%	13.61%	48.94%
ADISPEX+SF	2	43.13%	14.39%	62.78%	44.00%	15.16%	62.79%
	3	33.91%	5.85%	50.80%	40.48%	8.38%	50.78%
	4	24.61%	5.36%	49.32%	25.93%	5.96%	50.33%
DYCOSPEX	2	7.64%	-22.16%	24.82%	13.02%	-3.73%	5.30%
	3	21.12%	4.96%	28.97%	26.04%	6.57%	50.27%
	4	17.19%	1.53%	39.72%	23.15%	4.12%	39.62%
COCOSPEX	2	4.75%	-24.30%	20.75%	8.83%	-10.06%	21.45%
	3	8.49%	-0.06%	17.33%	11.28%	-0.02%	17.60%
	4	2.94%	-7.74%	11.04%	2.88%	-9.74%	11.06%

Table 5.2: Summary of Disabling Speculative Execution Techniques

The failings of each of the techniques presented in this chapter show the significance of speculative execution even with the TLP that can be extracted from an SMT machine. None of these techniques are able to hide enough latency even with support for 4 thread contexts and do not appear to be able to do so within an extremely aggressive SMT design.

Making the techniques more aggressive improved performance but did not make any of these methods better than the baseline. For some mixes improvements could be shown but it still remains unsuitable for a general purpose processor. Clearly, the ILP exploited by speculative execution is extremely important regardless of the amount of TLP that can be extracted among a reasonable SMT design. Future exploration can consider aggressive SMT designs as well as more intelligent instruction fetching logic to find ways to relax the requirement of speculative execution.

Chapter 6

Load-Hit Speculation

Another method to improve processor performance is speculation of a load hitting into the L1 cache. Because of the variable latency of load instructions this exploits the fact that most loads hit into the L1 cache [TB 01] and can be handled quickly and effectively. However, guarantees can not be made on the performance of the L1 cache. The size of the L1 cache determines the potential for cache misses and increasing the size of the cache is exponential in terms of hardware real estate.

Instead, a predictor (generally a branch predictor will do) is applied that uses prior cache history and prior load hit/miss history to speculate if the load will hit or miss. When the prediction is incorrect logic to replay the affected instructions and fix the pipeline is required. The costs of not using load-hit speculation can be seen in Figure 6.1.

Figure 6.1 shows the performance of disabling speculative scheduling with various issue-to-execute latencies (from 1 to 3 cycles) on a single-threaded machine. Performances are shown in Table 6.1 for the results shown in Figure 6.1. 3 cycle issue-to-execute is considered the default pipeline depth.

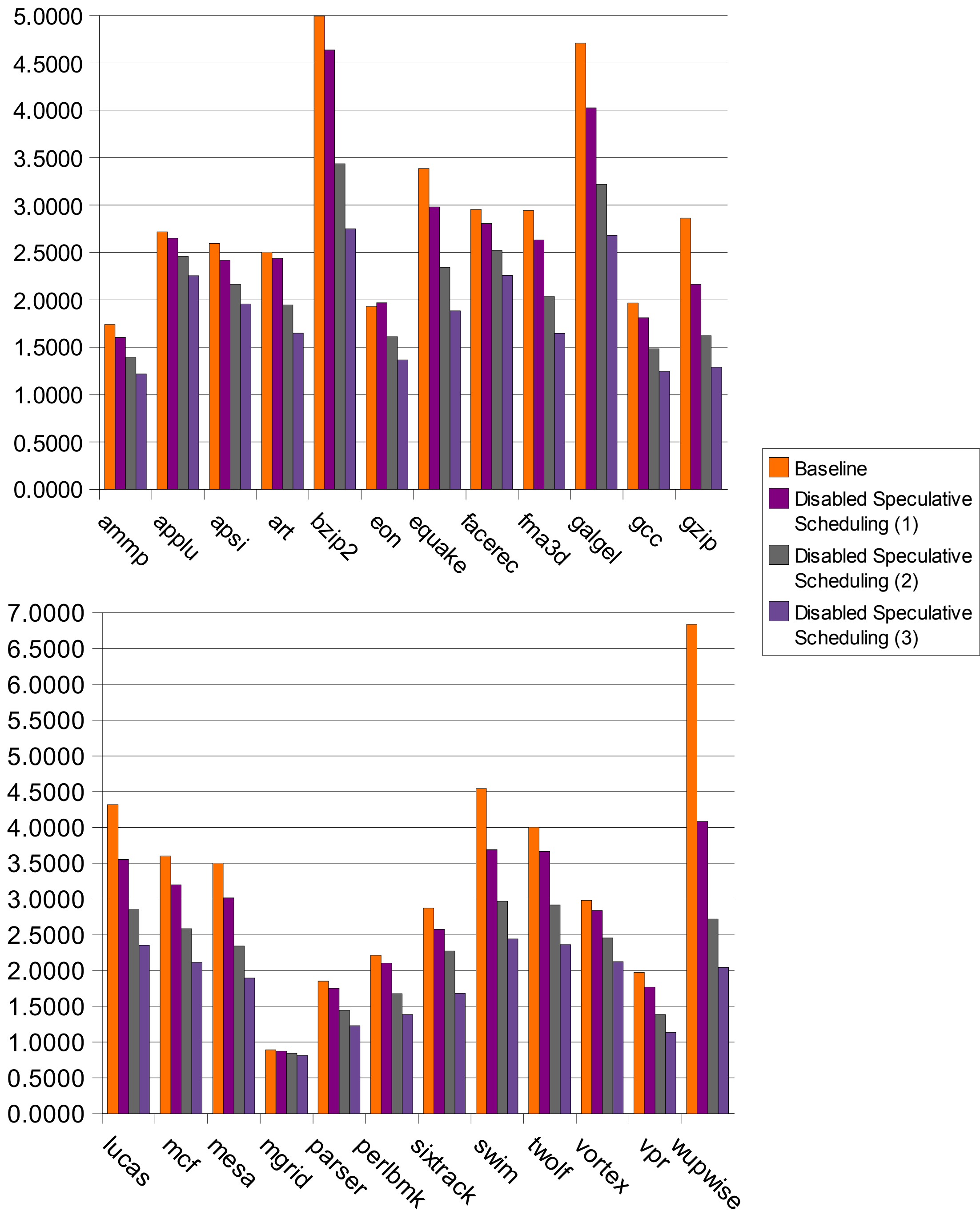


Figure 6.1: Single Thread Results for Disabled Speculative Scheduling

Issue-to-execute latency	Average Performance Loss	Min. Performance Loss	Max. Performance Loss
1 Cycle	10.30%	eon, -1.94%	wupwise, 40.28%
2 Cycles	26.22%	mgrid, 5.24%	wupwise, 60.18%
3 Cycles	37.90%	mgrid, 8.63%	wupwise, 70.14%

Table 6.1: Summary of Disabled Speculative Scheduling on Superscalar

Load-hit speculation allows speculative scheduling which permits bypassing of the bubbles created by issue-to-execute delay. Reduced delay obviously makes the impact of load-hit speculation less profound. Upon reaching execution the readiness of a load instruction is verified and unless there is a cache miss everything runs smoothly as shown in Figure 6.2. A cache miss requires that the load and all dependent instructions be rescheduled (Figure 6.3) with an unspecified pipeline bubble represented as N cycles. Removing load-hit speculation prevents loads from entering the execute stage until their resources are ready; eliminating the need for rescheduling (Figure 6.4) but forcing all dependent instructions to wait for the resolution of their respective load instruction.

Load Inst.	**W/S**	**RF**	**EX**	**WB**	
Dependent Inst.		**W/S**	**RF**	**EX**	**WB**
Cycle:	**t**	**t+1**	**t+2**	**t+3**	**t+4**

Figure 6.2: Pipeline Diagram for Correct Load-Hit Prediction

Load Inst.	**W/S**	**RF**	**EX**	**Miss**		**Waiting**	**W/S**	**RF**	**EX**
Dependent Inst.		**W/S**	**RF**	**Rescheduled**				**W/S**	**RF**
Cycle:	**t**	**t+1**	**t+2**	**t+3**		**t+N**	**t+N+1**	**t+N+2**	**t+N+3**

Figure 6.3: Pipeline Diagram for Incorrect Load-Hit Prediction

Load Inst.	**W/S**	**RF**	**EX**	**WB**		
Dependent Inst.				**W/S**	**RF**	**EX**
Cycle:	**t**	**t+1**	**t+2**	**t+3**	**t+4**	**t+5**

Figure 6.4: Execution Schedule Without Load-Hit Speculation

Load-hit speculation permits increased pipeline depth/complexity with reduced penalties on the datapath. Reduced issue-to-execute delay indicates a less complex pipeline and demonstrates that load-hit speculation is less valuable in this case. Increased issue-to-execute delay creates pipeline bubbles that are detrimental to the performance of the datapath, these are the situations that load-hit speculation proves to be valuable to exploiting instruction-level parallelism. A simpler pipeline may be able to sacrifice load-hit speculation in order to recover rescheduling and prediction hardware.

6.1 Disabling Speculative Scheduling: The Implementation

Disabling the load-hit predictor is accomplished by removing it from the pipeline. When a load instruction checks if its source registers are ready it does not use the load-hit predictor and instead checks the valid bit of the physical register file. No replays occur in this case, but the back-to-back execution of loads and load-dependent instructions is not possible as shown in Figure 6.4. This method will be examined for various issue-to-execute delays (1-3 cycles, 3 is the baseline delay).

6.2 Results of Disabling Speculative Scheduling on SMT

The results demonstrate that thread-level parallelism cannot cover up the pipeline bubbles created by a lack of speculative scheduling in a deep-schedule-to-execute pipeline. However, a simple pipeline (1 cycle issue-to-execute delay) small penalties over the baseline case (3 cycle issue-to-execute delay) that decrease with increased thread-level parallelism.

Figure 6.5 presents the performance of disabling speculative scheduling for issue-to-execute delays of 1, 2 and 3 cycles as throughput IPC for 2-threaded workloads. Without load-hit speculation the performance losses average 5.73%, 20.74% and 33.68% for issue-to-execute delays of 1, 2 and 3 respectively. For an issue-to-execute delay of 1 cycle performance loss ranges from -1.64% for Mix 2-1 to 10.71% for Mix 2.2. For an issue-to-execute delay of 2 cycle performance loss ranges from 11.67% for Mix 2-1 to 26.15% for Mix 2.6. For an issue-to-execute delay of 3 cycle performance loss ranges from 24.76% for Mix 2-1 to 40.87% for Mix 2.5.

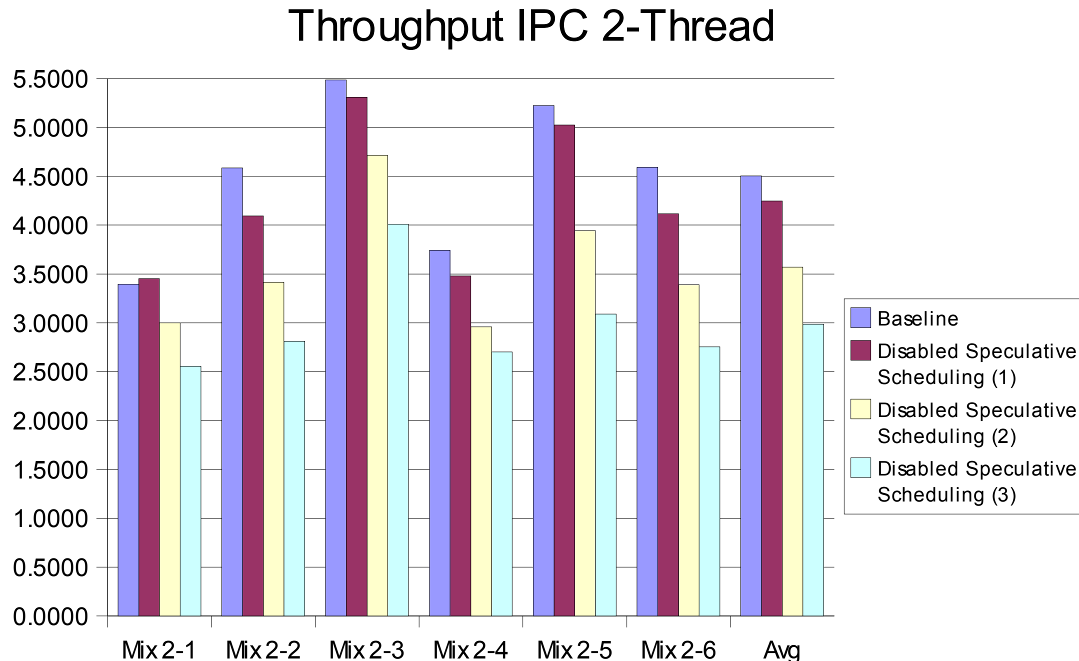


Figure 6.5: Throughput IPC for 2-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays

Figure 6.6 presents the performance of disabling speculative scheduling for issue-to-execute delays of 1, 2 and 3 cycles as throughput IPC for 3-threaded workloads. Without load-hit speculation the performance losses average 1.8%, 13.17% and 25.5% for issue-to-execute delays of 1, 2 and 3 respectively. For an issue-to-execute delay of 1 cycle performance loss ranges from -0.46% for Mix 3-5 to 5.06% for Mix 3.10. For an issue-to-

execute delay of 2 cycle performance loss ranges from 6.11% for Mix 3-6 to 19.88% for Mix 3.10. For an issue-to-execute delay of 3 cycle performance loss ranges from 16.13% for Mix 3-6 to 33.78% for Mix 3.10.

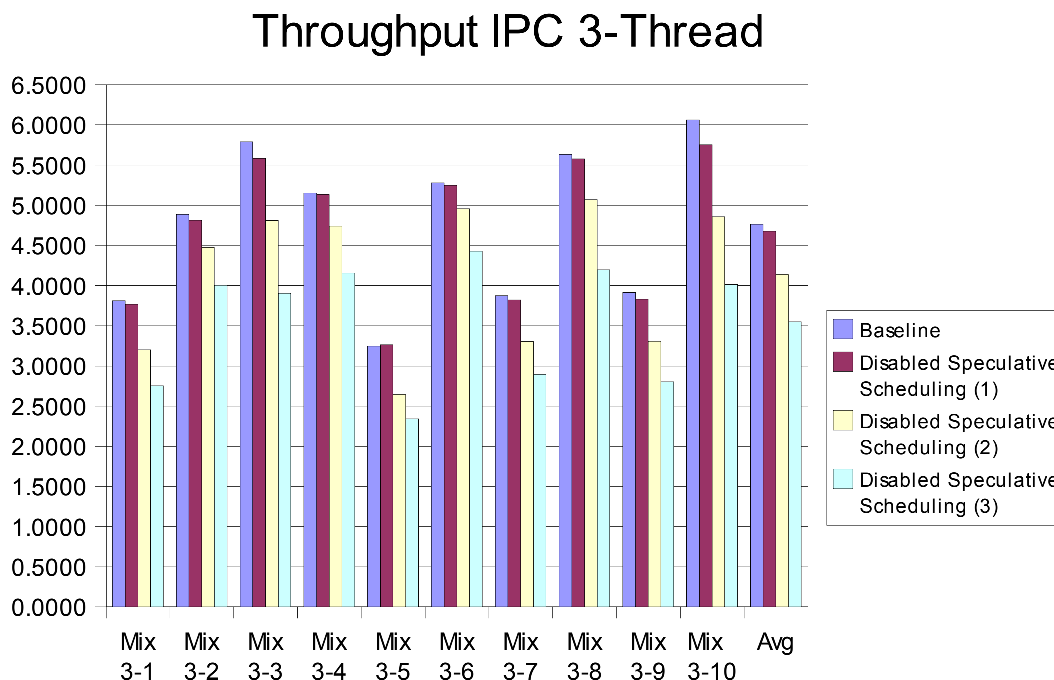


Figure 6.6: Throughput IPC for 3-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays

Figure 6.7 presents the performance of disabling speculative scheduling for issue-to-execute delays of 1, 2 and 3 cycles as throughput IPC for 4-threaded workloads. Without load-hit speculation the performance losses average 0.42%, 9.61% and 22.21% for issue-to-execute delays of 1, 2 and 3 respectively. For an issue-to-execute delay of 1 cycle performance loss ranges from -0.84% for Mix 4-9 to 1.44% for Mix 4.1. For an issue-to-execute delay of 2 cycle performance loss ranges from 4.49% for Mix 4-4 to 19.64% for Mix 4.1. For an issue-to-execute delay of 3 cycle performance loss ranges from 15.52% for Mix 4-3 to 29.89% for Mix 4.7.

These results show how exploiting thread-level parallelism hides pipeline bubbles generated by a lack of load-hit speculation. It also demonstrates the effect of pipeline

depth on the performance of the datapath.

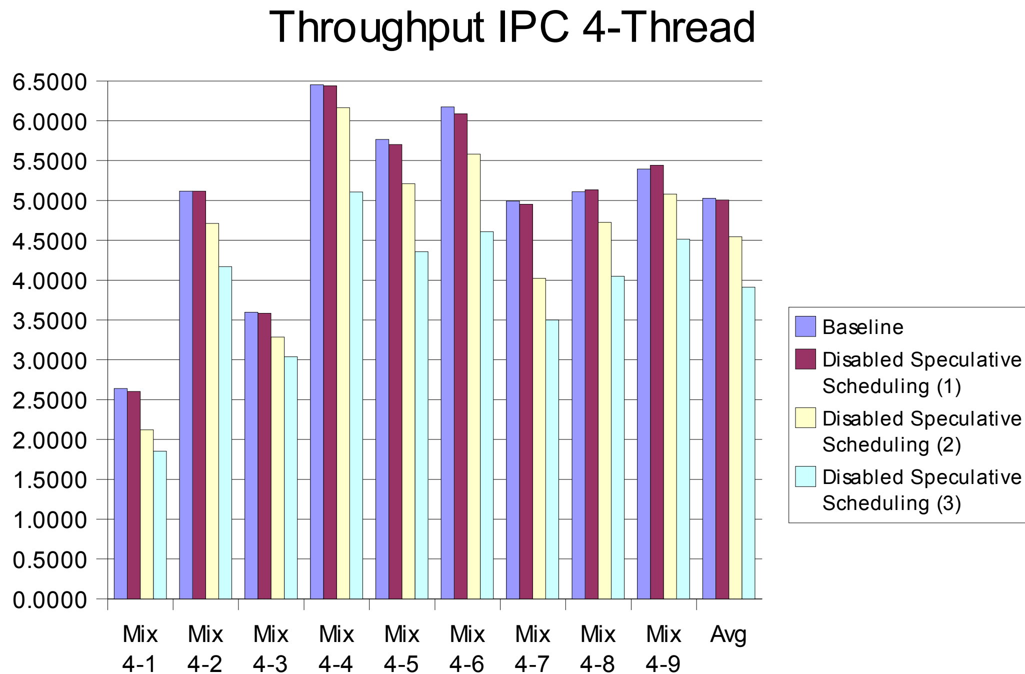


Figure 6.7: Throughput IPC for 4-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays

Figure 6.8 presents the performance of disabling speculative scheduling for issue-to-execute delays of 1, 2 and 3 cycles as fairness for 2-threaded workloads. Without load-hit speculation the fairness losses average 5.46%, 20.28% and 32.93% for issue-to-execute delays of 1, 2 and 3 respectively. For an issue-to-execute delay of 1 cycle fairness loss ranges from -1.63% for Mix 2-1 to 10.55% for Mix 2.2. For an issue-to-execute delay of 2 cycle fairness loss ranges from 11.67% for Mix 2-1 to 26.46% for Mix 2.6. For an issue-to-execute delay of 3 cycle fairness loss ranges from 24.77% for Mix 2-1 to 40.22% for Mix 2.5 and Mix 2.6.

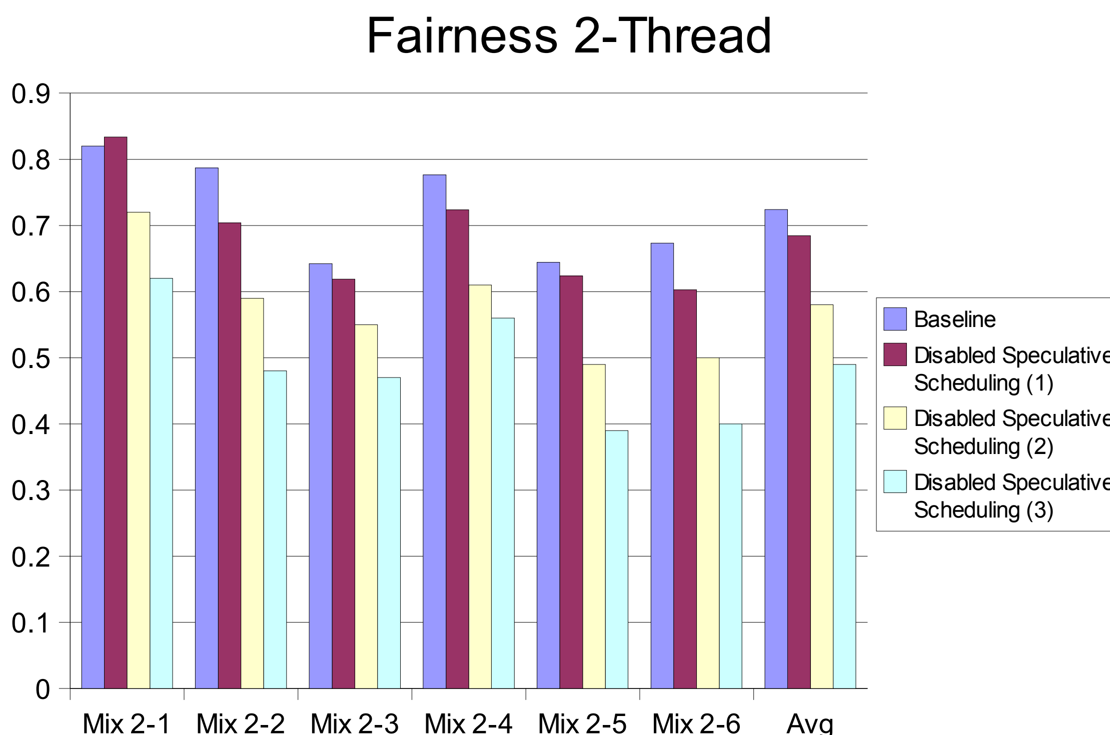


Figure 6.8: Fairness for 2-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays

Figure 6.9 presents the performance of disabling speculative scheduling for issue-to-execute delays of 1, 2 and 3 cycles as fairness for 3-threaded workloads. Without load-hit speculation the fairness losses average 1.68%, 13.73% and 26.08% for issue-to-execute delays of 1, 2 and 3 respectively. For an issue-to-execute delay of 1 cycle fairness loss ranges from -0.94% for Mix 3-5 to 5.45% for Mix 3.10. For an issue-to-execute delay of 2 cycle fairness loss ranges from 5.91% for Mix 3-6 to 20.81% for Mix 3.10. For an issue-to-execute delay of 3 cycle fairness loss ranges from 16.17% for Mix 3-6 to 34.61% for Mix 3.10.

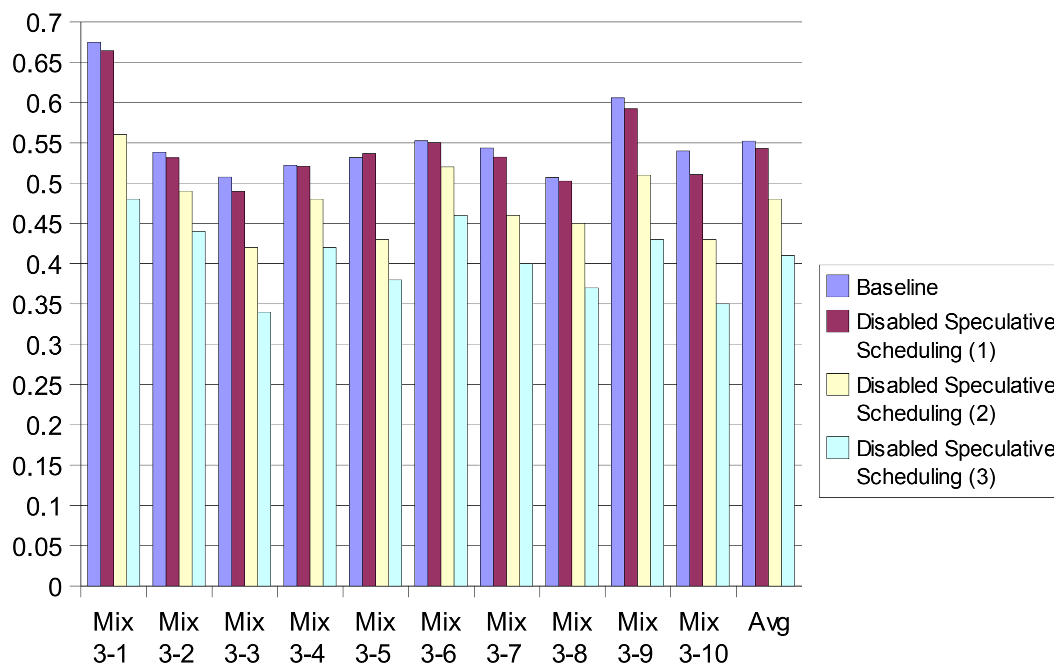


Figure 6.9: Fairness for 3-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays

Figure 6.10 presents the performance of disabling speculative scheduling for issue-to-execute delays of 1, 2 and 3 cycles as fairness for 4-threaded workloads. Without load-hit speculation the fairness losses average 0.34%, 10.51% and 22.64% for issue-to-execute delays of 1, 2 and 3 respectively. For an issue-to-execute delay of 1 cycle fairness loss ranges from -1.11% for Mix 4-9 to 1.42% for Mix 4.6. For an issue-to-execute delay of 2 cycle fairness loss ranges from 4.62% for Mix 4-4 to 20.91% for Mix 4.7. For an issue-to-execute delay of 3 cycle fairness loss ranges from 15.6% for Mix 4-2 to 31.46% for Mix 4.7.

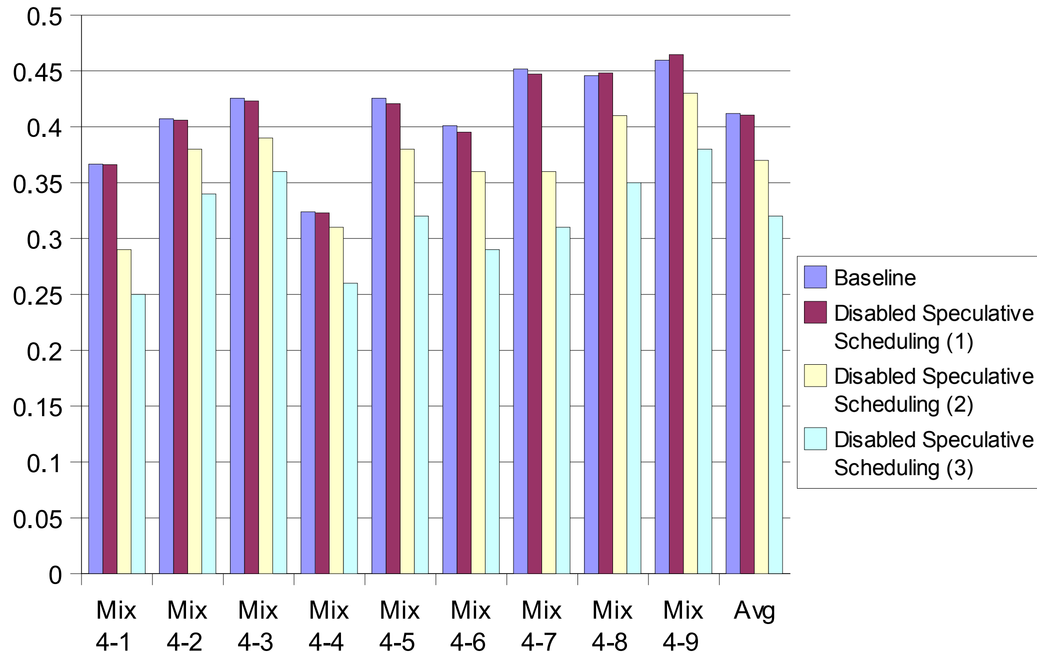


Figure 6.10: Fairness for 4-threaded Workloads of Disabled Speculative Scheduling with Various Issue-to-Execute Delays

6.3 Summary of Disabling Speculative Scheduling on SMT

Disabled Load-Hit Speculation with...	# Threads	Average IPC Performance Loss	Min. IPC Performance Loss	Max. IPC Performance Loss	Average Fairness Loss	Min. Fairness Loss	Max. Fairness Loss
Issue-to-execute delay of 1 cycle	2	5.73%	-1.64%	10.71%	5.46%	-1.63%	10.55%
	3	1.80%	-0.46%	5.06%	1.68%	-0.94%	5.45%
	4	0.42%	-0.84%	1.44%	0.34%	-1.11%	1.42%
Issue-to-execute delay of 2 cycle	2	20.74%	11.67%	26.15%	20.28%	11.67%	26.46%
	3	13.17%	6.11%	19.88%	13.73%	5.91%	20.81%
	4	9.61%	4.49%	19.64%	10.51%	4.62%	20.91%
Issue-to-execute delay of 3 cycle (Baseline)	2	33.68%	24.76%	40.87%	32.93%	24.77%	40.22%
	3	25.50%	16.13%	33.78%	26.08%	16.17%	34.61%
	4	22.21%	15.52%	29.89%	22.64%	15.60%	31.46%

Table 6.2: Summary of Disabled Speculative Scheduling on SMT

The results indicate that removing load-hit speculation can be done at a reasonable cost if the pipeline depth can be reduced as well. Thread-level parallelism is effective at hiding latency generated by a lack of load-hit speculation but even with a 4-way SMT machine

performance losses can average 22.21%. Further aggressive designs may be suitable for reducing the need for load-hit speculation but need to be examined more completely.

Chapter 7

Relaxing Scheduling Loops on SMT Pipelined Selection Logic

Back-to-Back execution allows a processor to exploit instruction-level parallelism by allowing dependent instructions to execute immediately after the instruction they are depending on executes. When an instruction is selected it can immediately broadcast its instruction tags to wakeup all appropriate waiting instructions to be issued for just-in-time access via the result bypass network; allowing these registers to be read before writeback. However, Back-to-Back execution is becoming difficult to maintain with current microprocessing trends increasing the burden on, an already tight, pipeline loop.

Figure 7.1 represents a pipeline diagram with Back-to-Back execution with three dependent, single-cycle, instructions in-flight. As additional stresses force the pipelining of the Wakeup/Select logic the performance result and penalty is shown in Figure 7.2.

I1	**W/S**	**RF**	**EX**	**WB**		
I2		**W/S**	**RF**	**EX**	**WB**	
I3			**W/S**	**RF**	**EX**	**WB**
Cycle:	**t**	**t+1**	**t+2**	**t+3**	**t+4**	**t+5**

Figure 7.1: Pipeline Diagram with Back-to-Back Execution

I1	W	S	RF	EX	WB				
I2			W	S	RF	EX	WB		
I3					W	S	RF	EX	WB
Cycle:	t	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8

Figure 7.2: Pipeline Diagram with No Back-to-Back Execution

Pipelining Wakeup/Select is an obvious performance problem. Figure 7.2 clearly shows that each instruction is delayed by one cycle more than the previous instruction was delayed. A modern general purpose processor can potentially execute billions of instructions each cycle and incremental delays like these are undesirable.

These penalties can be offset by applying thread-level parallelism to fill in these pipeline bubbles, shown in Figure 7.3. With a conservative design, 2-way SMT, a second thread can fill in the pipeline bubble generated by pipelining Wakeup/Select. The first thread doesn't complete its three instruction as fast as in Figure 7.1 but cycles are not wasted doing nothing, instead performing work for a different thread during the same time period.

I1	W	S	RF	EX	WB				
Thread 2: I1		W	S	RF	EX	WB			
I2			W	S	RF	EX	WB		
Thread 2: I2				W	S	RF	EX	WB	
I3					W	S	RF	EX	WB
Cycle:	t	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8

Figure 7.3: Pipeline Diagram (2-way SMT) with No Back-to-Back Execution

For a single-threaded machine pipelining Wakeup/Select into two cycles causes an average performance loss of 9.93% ranging from 1.52% for mgrid and 31.88% for wupwise (see Figure 7.4).

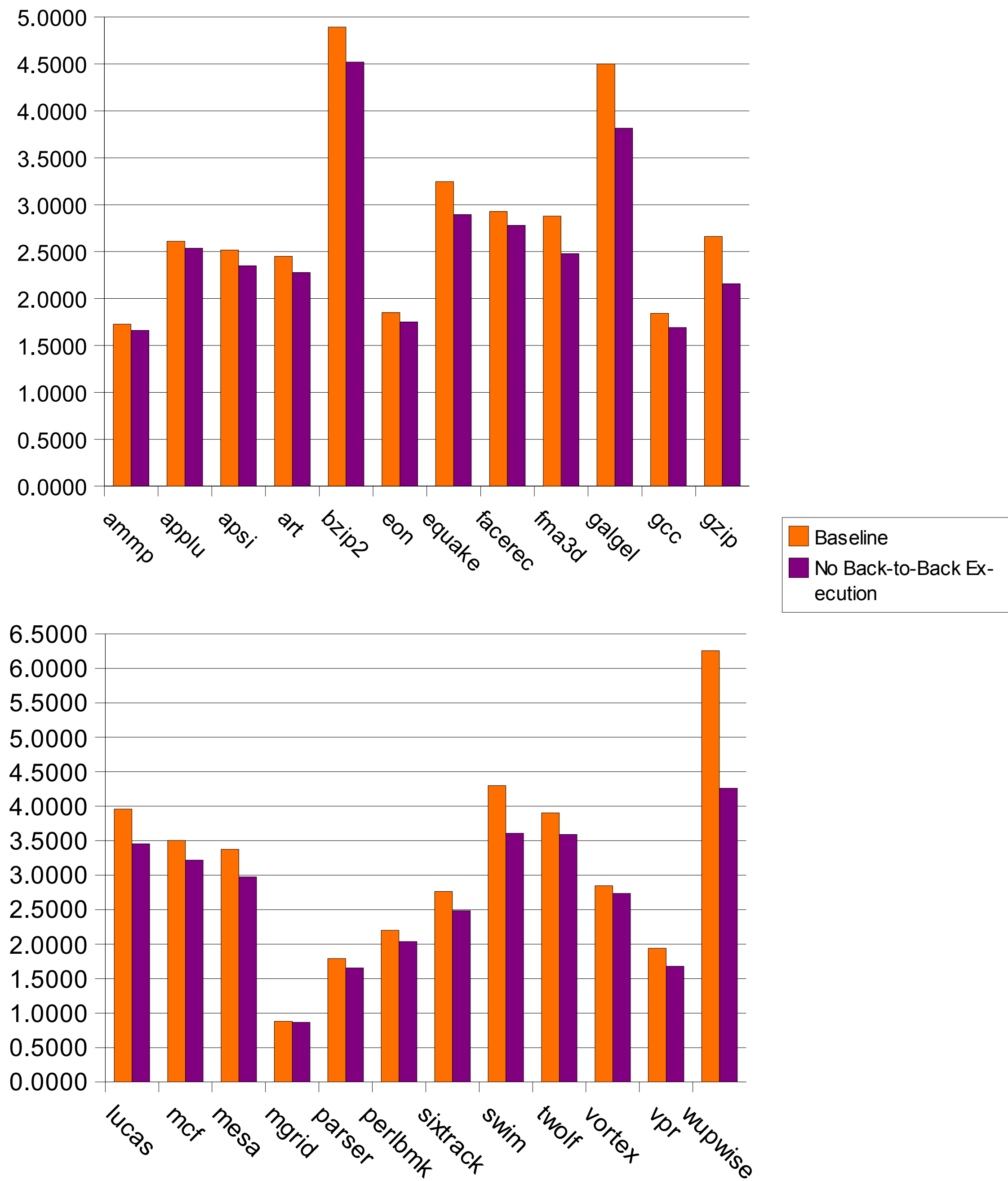


Figure 7.4: Disabling Back-to-Back Execution on a Single Threaded Processor

7.1 Pipelining the Scheduling Logic Over 2 Cycles: The Implementation

Disabling Back-to-Back execution is achieved by not allowing instructions to broadcast their tags when they are selected for execution. These broadcasts normally allow the scheduler to issue dependent instructions that would be able to pick up their source operands just-in-time. Without the broadcasts back-to-back execution is not possible and dependent instructions cannot be issued is successive cycles. Relaxing of this pipeline loop allows the implementation of more advanced selection logic to further improve performance.

7.2 Results of Pipelining the Scheduling Logic Over 2 Cycles

For this implementation the effect of pipelining scheduling logic on SMT is examined. Later in this chapter this will be examined in tandem with various selection policies. Position Based is selected as the representative baseline model because it is the simplest to implement and represents a reasonable single-cycle selection policy (all implementations in previous chapters used oldest first as their selection policy).

Removing Back-to-Back execution (the equivalent of pipelining Wakeup/Select over two cycles), impacts throughput IPC as well as fairness. The following results demonstrate that thread-level parallelism can hide the pipeline bubbles generated by a lack of Back-to-Back execution.

Figures 7.5-7.7 present the performance of disabling back-to-back execution (as described in section 7.1). The results are presented in terms of throughput IPC. For 2-threaded workloads (Figure 7.5) there is an average performance loss of 5.65% ranging

from 2.53% for Mix 2-3 to 7.81% for Mix 2-6. For 3-threaded workloads (Figure 7.6) there is an average performance loss of 4.71% ranging from 1.21% for Mix 3-6 to 11.95% for Mix 3-5. For 4-threaded workloads (Figure 7.7) there is an average performance loss of 3.47% ranging from 0.78% for Mix 4-4 to 6.86% for Mix 4-7.

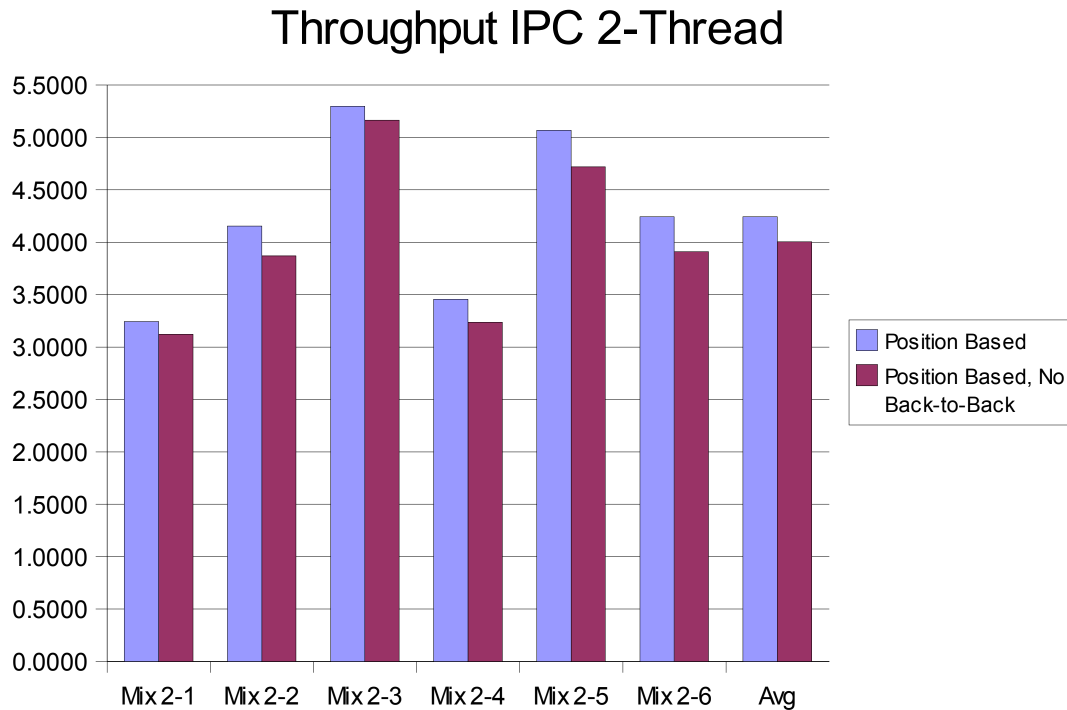


Figure 7.5: Throughput IPC for 2-threaded Workloads with Disabled Back-to-Back Execution

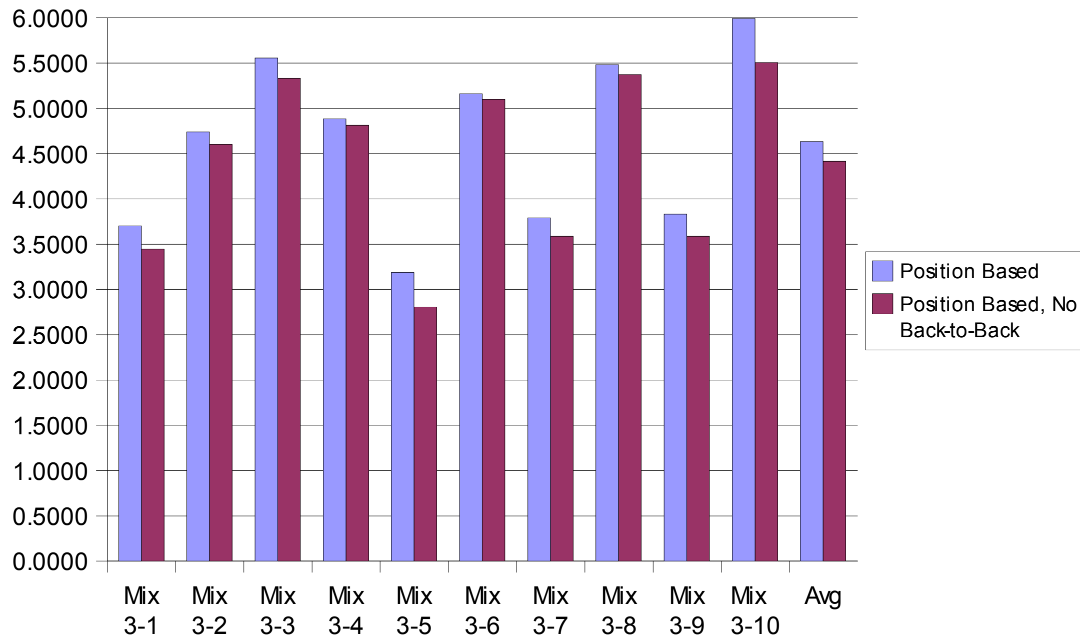


Figure 7.6: Throughput IPC for 3-threaded Workloads with Disabled Back-to-Back Execution

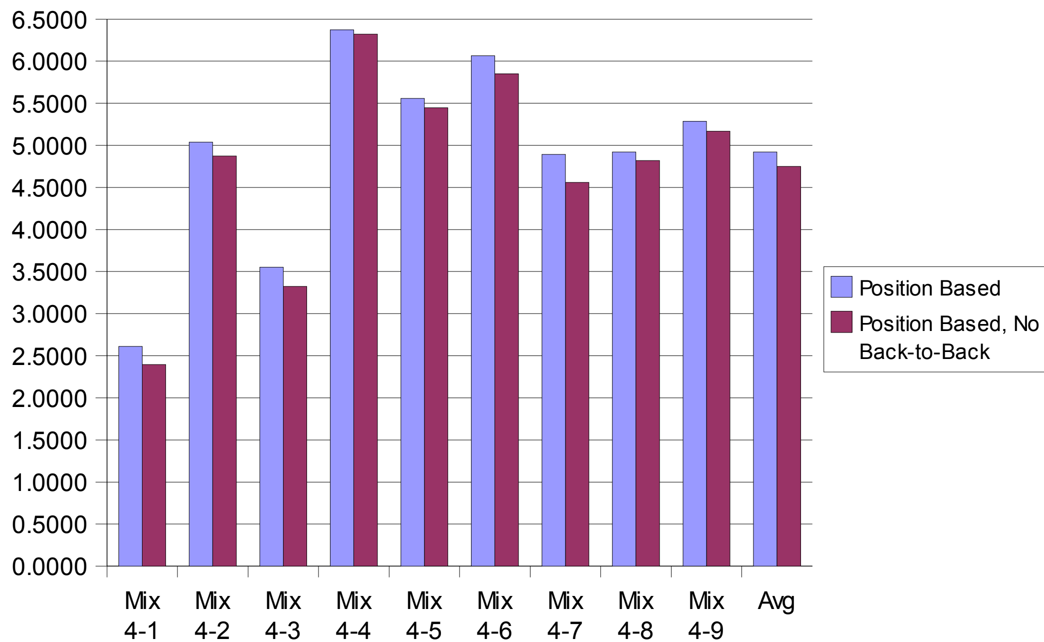


Figure 7.7: Throughput IPC for 4-threaded Workloads with Disabled Back-to-Back Execution

Figures 7.8-7.10 present the performance of disabling back-to-back execution (as described in section 7.1). The results are presented in terms of fairness. For 2-threaded workloads (Figure 7.8) there is an average fairness loss of 5.63% ranging from 2.41% for Mix 2-3 to 7.88% for Mix 2-6. For 3-threaded workloads (Figure 7.9) there is an average fairness loss of 5.23% ranging from 1.18% for Mix 3-6 to 11.57% for Mix 3-5. For 4-threaded workloads (Figure 7.10) there is an average fairness loss of 3.85% ranging from 0.78% for Mix 4-4 to 8.17% for Mix 4-1.

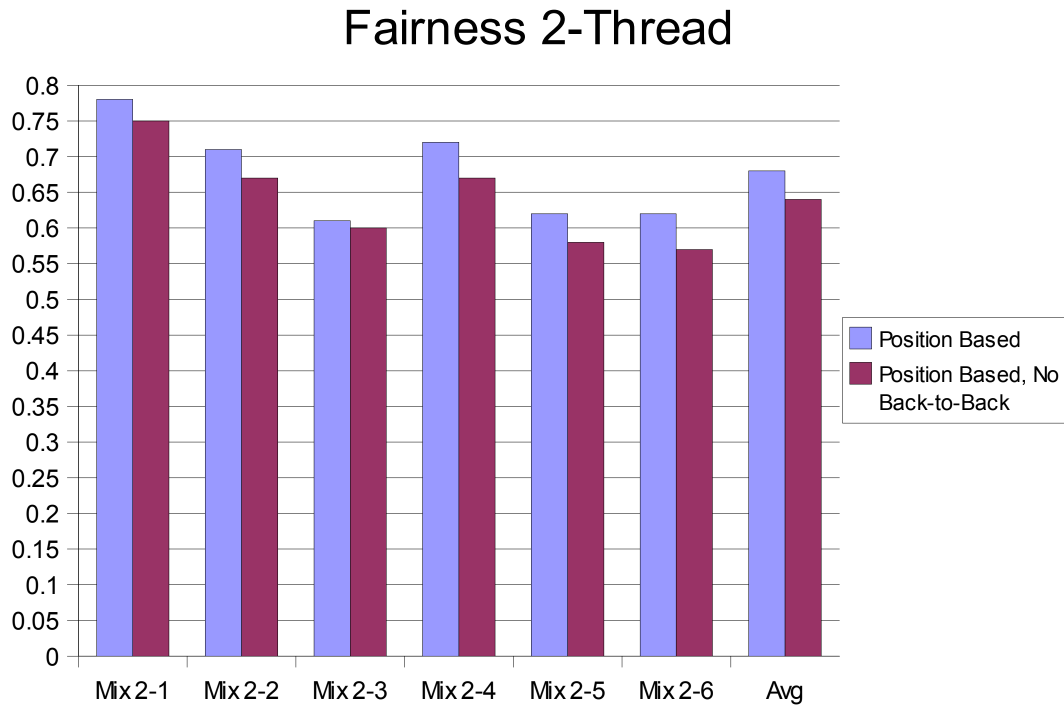


Figure 7.8: Fairness for 2-threaded Workloads with Disabled Back-to-Back Execution

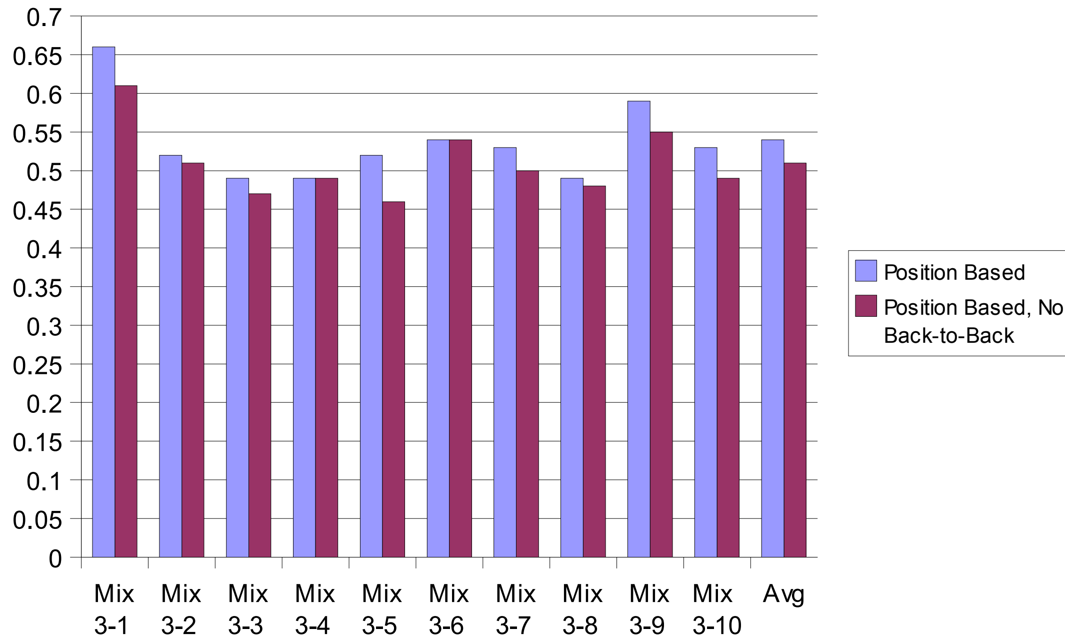


Figure 7.9: Fairness for 3-threaded Workloads with Disabled Back-to-Back Execution

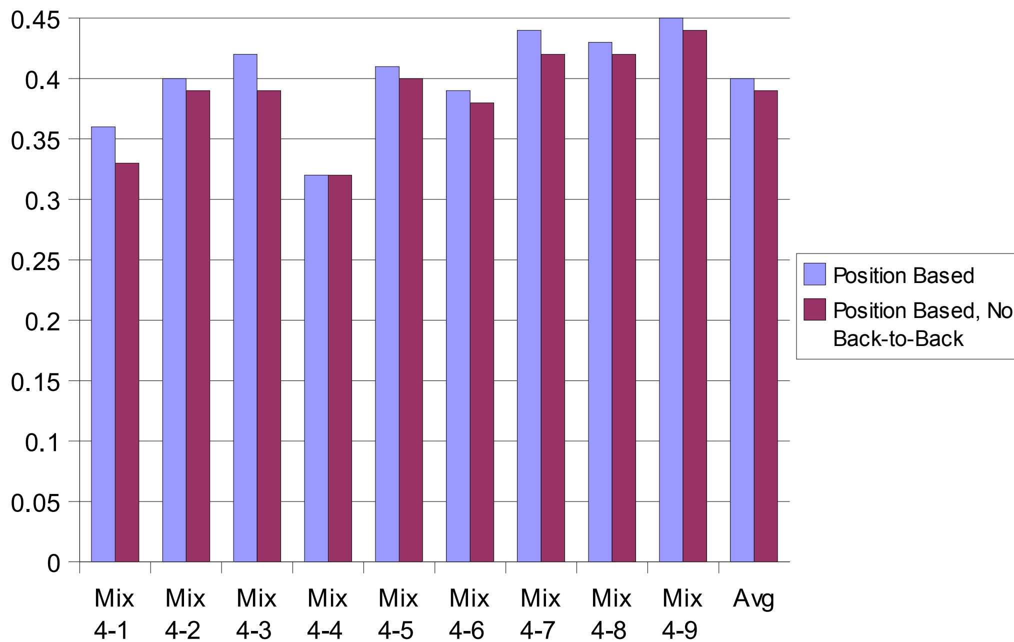


Figure 7.10: Fairness for 4-threaded Workloads with Disabled Back-to-Back Execution

7.3 Summary of Pipelining Scheduling Loops Over 2 Cycles

	# Threads	Average IPC Performance Loss	Min. IPC Performance Loss	Max. IPC Performance Loss	Average Fairness Loss	Min. Fairness Loss	Max. Fairness Loss
No Back-to-Back Execution	2	5.65%	2.53%	7.81%	5.63%	2.41%	7.88%
	3	4.71%	1.45%	11.95%	5.23%	1.18%	11.57%
	4	3.47%	0.78%	8.21%	3.85%	0.78%	8.17%

Table 7.1: Summary of Pipelining Scheduling Loops Over 2 Cycles on SMT

These results indicate that back-to-back execution is desirable but only provides a small benefit. More aggressive SMT designs reduce the benefit exploited by back-to-back execution and improved selection techniques with an aggressive SMT design may completely overcome losses incurred by removing back-to-back execution.

7.4 Selection Methods

Prior work by Butler and Patt [BP 92] indicated that performance was largely independent from the selection method that is implemented. However, this work applies only to superscalar and should be considered on an SMT machine. On SMT, improved selection methods may provide better performance and can more effectively utilize a pipelined Wakeup/Select mechanism. The following sections will examine three different selection methods, with pipelined and non-pipelined selection logic, in comparison to the Position Based method.

7.5 Implementation of Various Selection Methods

The baseline implementation, Position Based, causes instructions to be executed based on when they enter the issue queue. To do this, instructions that are dispatched are placed at the back of the queue and extracted from the front of the queue.

An Oldest First policy inserts dispatched instructions into the issue queue in program order. This is accomplished by scanning through the issue queue each time an instruction is dispatched and inserting it before a newer instruction from the same thread. The issue queue can then be read from front to back when extracting instructions for execution.

Distributed and One Thread Bias techniques are different from the first two methods due to additional logic requirements. Each thread maintains a *Ready_IQ_Count* counter that keeps track of the number of instructions from that thread that reside in the issue queue. *Ready_IQ_Count* is incremented each time an instruction is dispatched and decremented each time an instruction is selected for execution. A *Round_Robin* counter is used to determine which thread to begin selection from (this can also be implemented using modular arithmetic on an existing cycle counter). Finally, gatekeeper logic is implemented to perform the actual selection. The gatekeeper maintains a counter for each thread that indicates how many instructions it can select from a given thread, decrementing it each time selection is made as well as counter to indicate how many more instructions can be issued this cycle (*Issue_Capacity*). The gatekeeper scans through the issue queue selecting the appropriate number of instructions for execution.

One Thread Bias gives a bias to a particular thread each cycle determined using a round robin technique (accomplished using the *Round_Robin* counter). If the *Ready_IQ_Count* for that thread is greater, or equal, than the issue-width the gatekeeper is set to select a number instructions equal to the issue-width from that thread. Otherwise, it selects all the

available instructions for that thread, adjusts *Issue_Capacity* to represent the amount of issue-width remaining and moves to the next thread, repeating this until either no issue-width or no ready instructions remain.

A Distributed method attempts to evenly select from each of the threads while using all of the available issue-width. As long as some *Issue_Capacity* remains the gatekeeper is told to issue an instruction from each thread that has an instruction remaining. When more threads have an instruction available than the available *Issue_Capacity* threads are selected by using the *Round_Robin* counter; starting from the thread indicated by the counter and moving to the next thread until *Issue_Capacity* is zero.

Each of these selection methods is also examined in parallel with disabling Back-to-Back execution in order to represent the cost of pipelining each of these selection methods. The implementation of removing Back-to-Back execution is described in section 7.1.

7.6 Results of Various Selection Methods

The results will compare the different selection methods to the baseline method, (Position Based) including the combination of each method with disabled back-to-back execution.

Figure 7.11 presents the performance of each selection method (as described in section 7.5) for 2-threaded workloads. The results are presented as throughput IPC. Pipelined Position Based has an average performance loss of 5.65% ranging from 2.53% for Mix 2-3 to 7.81% for Mix 2-6. Distributed has an average performance loss of -5.56% ranging

from -8.38% for Mix 2-4 to -3.46 for Mix 2-5. Pipelined Distributed has an average performance loss of 2.4% ranging from -0.08% for Mix 2-1 to 4.3% for Mix 2-5. One Thread Bias has an average performance loss of -4.01% ranging from -6.43 for Mix 2-6 to -2.32 for Mix 2-5. Pipelined One Thread Bias has an average performance loss of 3.18% ranging from 1.01% for Mix 2-3 to 5.12% for Mix 2-2. Oldest First has an average performance loss of -6.13% ranging from -10.38% for Mix 2-2 to -3.07% for Mix 2-5. Pipelined Oldest First has an average performance loss of 1.61% ranging from -1.3% for Mix 2-2 to 4.64% for Mix 2-5.

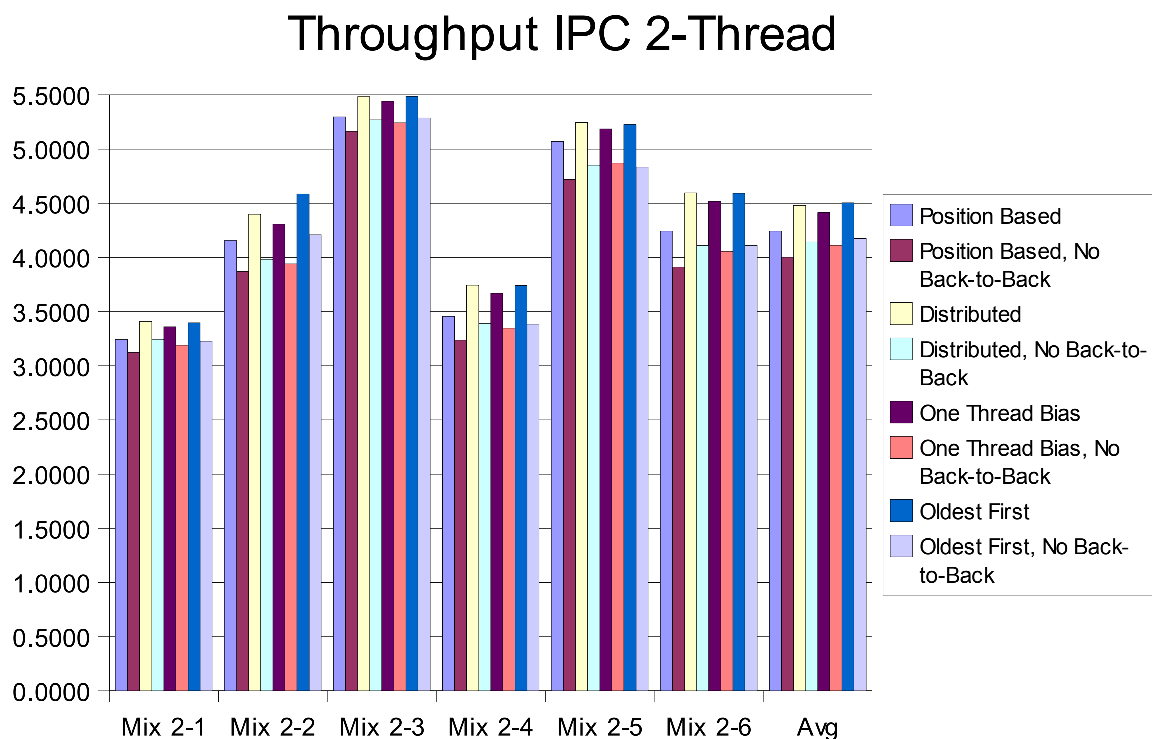


Figure 7.11: Throughput IPC for 2-threaded Workloads for Various Selection Methods

Figure 7.12 presents the performance of each selection method (as described in section 7.5) for 3-threaded workloads. The results are presented as throughput IPC. Pipelined Position Based has an average performance loss of 4.71% ranging from 1.21% for Mix 3-

6 to 11.95% for Mix 3-5. Distributed has an average performance loss of -2.96% ranging from -6.15% for Mix 3-4 to -0.38 for Mix 3-10. Pipelined Distributed has an average performance loss of 2.81% ranging from -2.84% for Mix 3-4 to 9.71% for Mix 3-5. One Thread Bias has an average performance loss of -1.32% ranging from -3.18 for Mix 3-4 to -0.38% for Mix 3-10. Pipelined One Thread Bias has an average performance loss of 4.16% ranging from -0.53% for Mix 3-4 to 11.36% for Mix 3-5. Oldest First has an average performance loss of -2.83% ranging from -5.48% for Mix 3-4 to -1.12% for Mix 3-10. Pipelined Oldest First has an average performance loss of 2.86% ranging from -2.34% for Mix 3-4 to 10.33% for Mix 3-5.

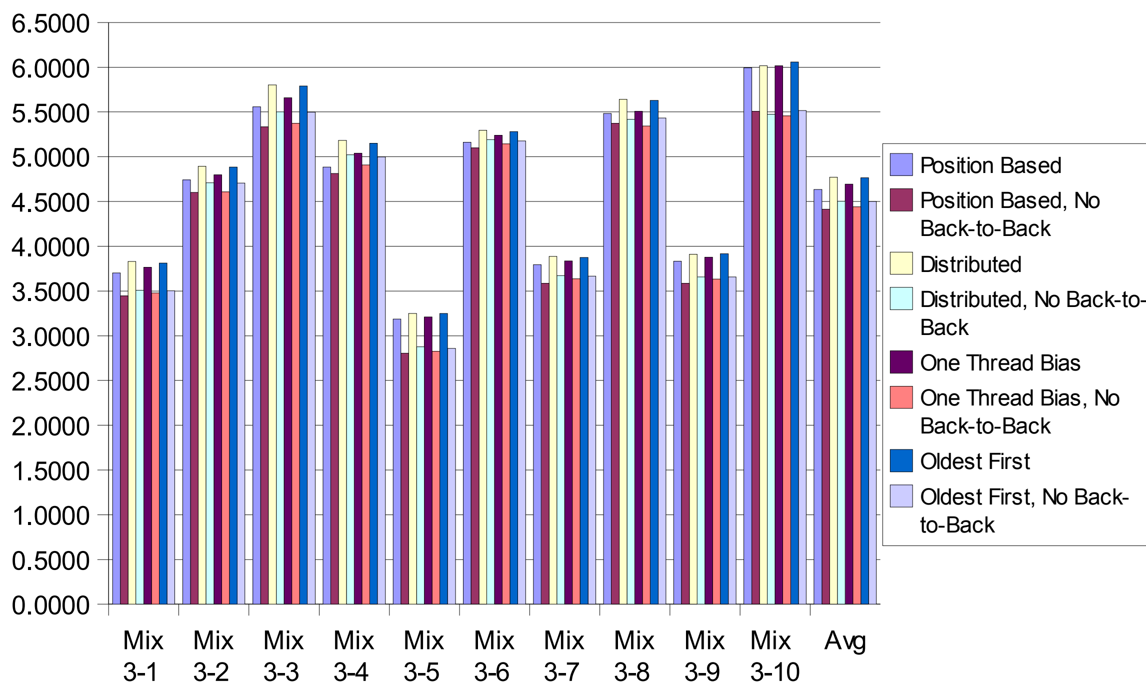


Figure 7.12: Throughput IPC for 3-threaded Workloads for Various Selection Methods

Figure 7.13 presents the performance of each selection method (as described in section 7.5) for 4-threaded workloads. The results are presented as throughput IPC. Pipelined

Position Based has an average performance loss of 3.74% ranging from 0.78% for Mix 4-4 to 8.21% for Mix 4-1. Distributed has an average performance loss of -2.6% ranging from -5.14% for Mix 4-8 to -1.45 for Mix 4-2. Pipelined Distributed has an average performance loss of 1.54% ranging from -1.71% for Mix 4-8 to 5.93% for Mix 4-1. One Thread Bias has an average performance loss of -1.59% ranging from -3.29 for Mix 4-8 to -0.29% for Mix 4-2. Pipelined One Thread Bias has an average performance loss of 2.84% ranging from 0.52% for Mix 4-8 to 6.46% for Mix 4-1. Oldest First has an average performance loss of -2.15% ranging from -3.85% for Mix 4-8 to -1.18% for Mix 4-1. Pipelined Oldest First has an average performance loss of 1.76% ranging from -1% for Mix 4-8 to 5.91% for Mix 4-1.

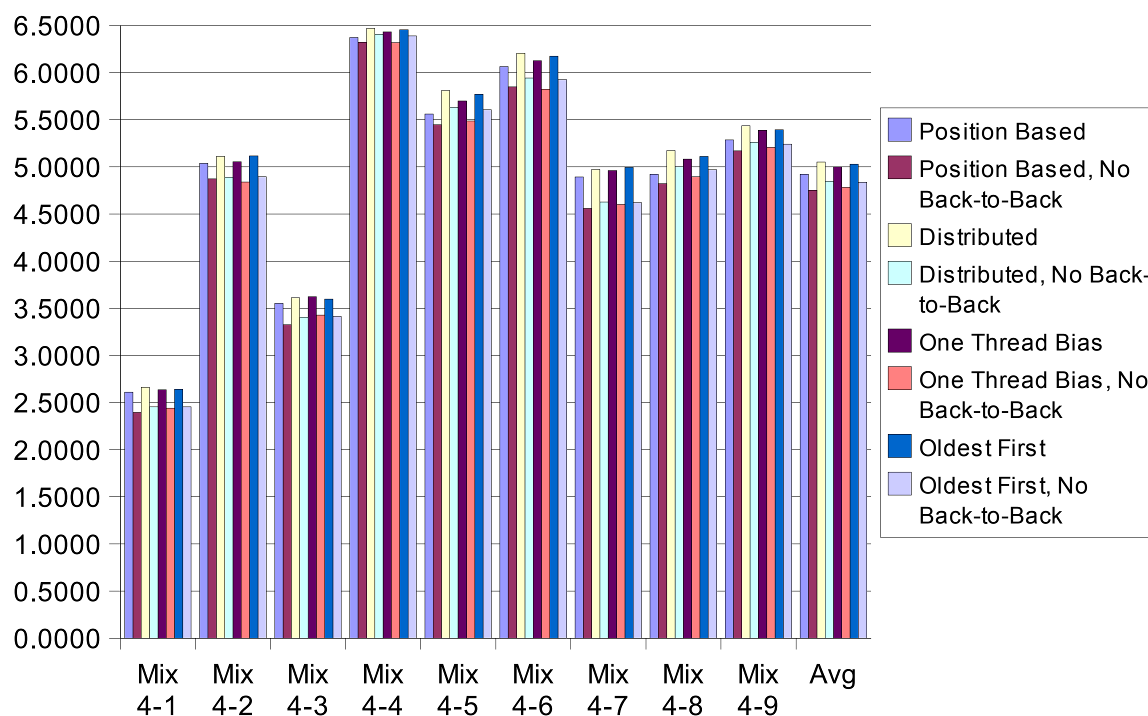


Figure 7.13: Throughput IPC for 4-threaded Workloads for Various Selection Methods

Figure 7.14 presents the performance of each selection method (as described in section

7.5) for 2-threaded workloads. The results are presented as fairness. Pipelined Position Based has an average fairness loss of 5.63% ranging from 2.41% for Mix 2-3 to 7.88% for Mix 2-6. Distributed has an average fairness loss of -5.99% ranging from -8.54% for Mix 2-6 to -3.39 for Mix 2-5. Pipelined Distributed has an average fairness loss of 2.14% ranging from -0.13% for Mix 2-3 to 4.32% for Mix 2-5. One Thread Bias has an average fairness loss of -4.34% ranging from -6.54 for Mix 2-6 to -3.65 for Mix 2-1. Pipelined One Thread Bias has an average fairness loss of 3.12% ranging from 0.47% for Mix 2-3 to 5.14% for Mix 2-2. Oldest First has an average fairness loss of -6.6% ranging from -10.16% for Mix 2-2 to -3.15% for Mix 2-5. Pipelined Oldest First has an average fairness loss of 1.33% ranging from -1.13% for Mix 2-2 to 4.55% for Mix 2-5.

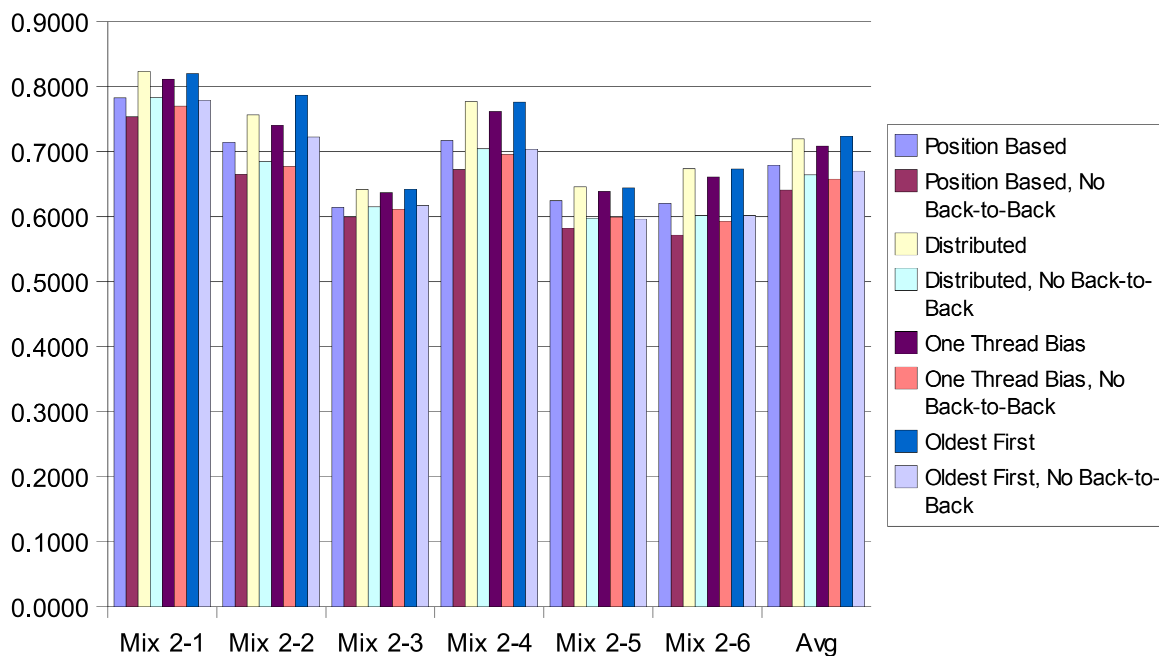


Figure 7.14: Fairness for 2-threaded Workloads for Various Selection Methods

Figure 7.15 presents the performance of each selection method (as described in section 7.5) for 3-threaded workloads. The results are presented as fairness. Pipelined Position

Based has an average fairness loss of 5.23% ranging from 1.18% for Mix 3-6 to 11.57% for Mix 3-5. Distributed has an average fairness loss of -3.02% ranging from -6.25% for Mix 3-4 to -0.28 for Mix 3-10. Pipelined Distributed has an average fairness loss of 3.24% ranging from -2.98% for Mix 3-4 to 9.38% for Mix 3-10. One Thread Bias has an average fairness loss of -1.34% ranging from -3.18 for Mix 3-4 to -0.37% for Mix 3-10. Pipelined One Thread Bias has an average fairness loss of 4.63% ranging from -0.61% for Mix 3-4 to 10.73% for Mix 3-5. Oldest First has an average fairness loss of -2.79% ranging from -5.65% for Mix 3-4 to -1.05% for Mix 3-10. Pipelined Oldest First has an average fairness loss of 3.41% ranging from -2.49% for Mix 3-4 to 9.59% for Mix 3-5.

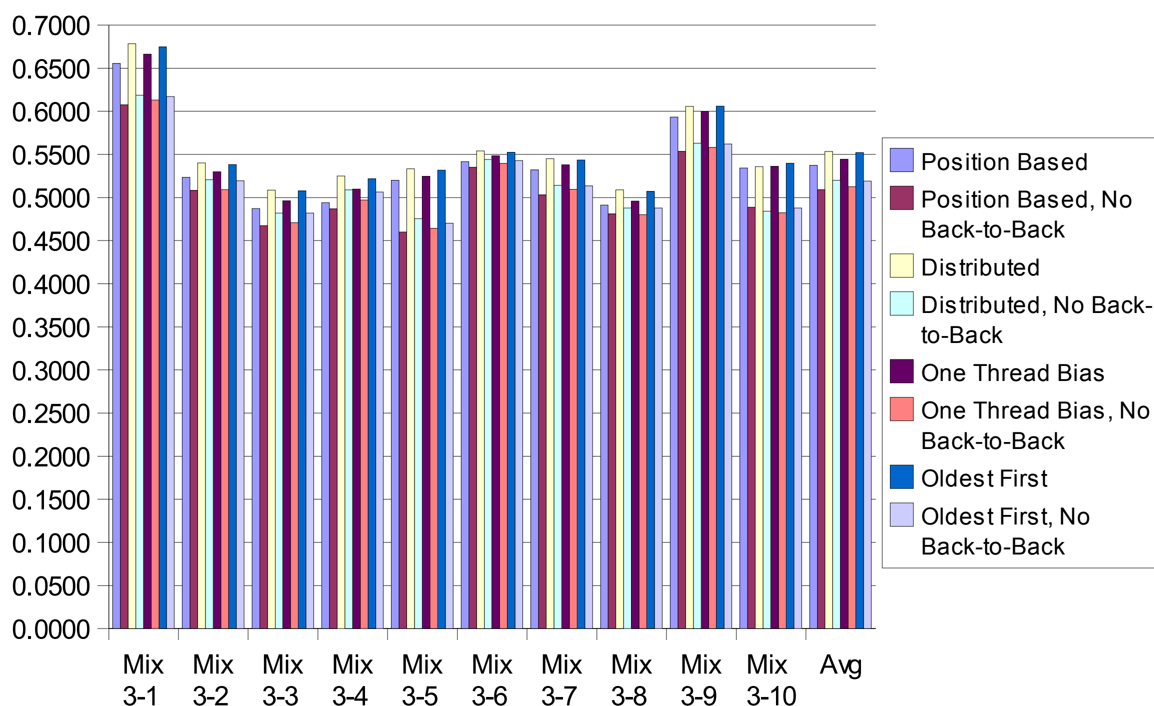


Figure 7.15: Fairness for 3-threaded Workloads for Various Selection Methods

Figure 7.16 presents the performance of each selection method (as described in section 7.5) for 4-threaded workloads. The results are presented as fairness. Pipelined Position

Based has an average fairness loss of 3.85% ranging from 0.78% for Mix 4-4 to 8.17% for Mix 4-1. Distributed has an average fairness loss of -2.51% ranging from -4.93% for Mix 4-8 to -1.13 for Mix 4-7. Pipelined Distributed has an average fairness loss of 1.89% ranging from -1.58% for Mix 4-8 to 6.74% for Mix 4-1. One Thread Bias has an average fairness loss of -1.61% ranging from -3.21 for Mix 4-8 to -0.40% for Mix 4-2. Pipelined One Thread Bias has an average fairness loss of 3.08% ranging from 0.54% for Mix 4-8 to 6.88% for Mix 4-1. Oldest First has an average fairness loss of -2.16% ranging from -3.75% for Mix 4-8 to -0.85% for Mix 4-1. Pipelined Oldest First has an average fairness loss of 2.01% ranging from -0.93% for Mix 4-8 to 6.27% for Mix 4-1.

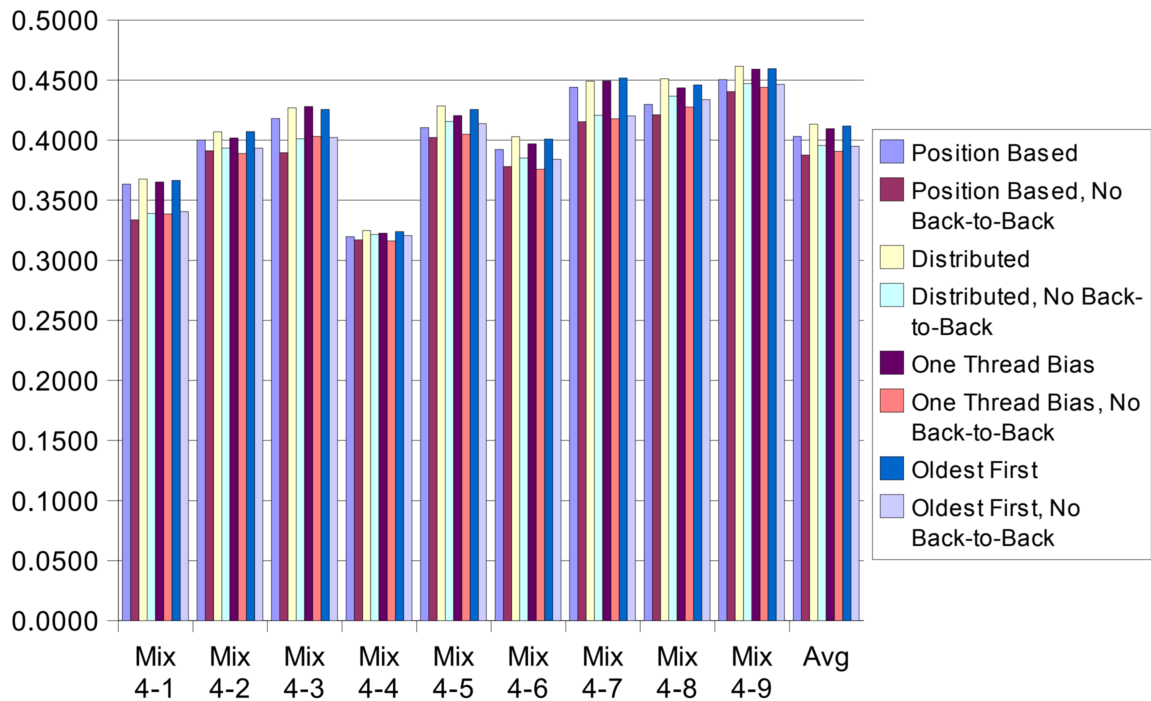


Figure 7.16: Fairness for 4-threaded Workloads for Various Selection Methods

7.7 Summary of Various Selection Methods

Selection Method	Back-to-Back	# Threads	Average IPC Performance Loss	Min. IPC Performance Loss	Max. IPC Performance Loss	Average Fairness Loss	Min. Fairness Loss	Max. Fairness Loss
Position Based	Yes	2, 3, 4	Baseline					
	No	1	9.93%	1.52%	31.88%			
		2	5.65%	2.53%	7.81%	5.63%	2.41%	7.88%
		3	4.71%	1.21%	11.95%	5.23%	1.18%	11.57%
		4	3.74%	0.78%	8.21%	3.85%	0.78%	8.17%
Distributed	Yes	2	-5.56%	-8.38%	-3.46%	-5.99%	-8.54%	-3.39%
		3	-2.96%	-6.15%	-0.38%	-3.02%	-6.25%	-0.28%
		4	-2.60%	-5.14%	-1.45%	-2.51%	-4.93%	-1.13%
	No	2	2.40%	0.08%	4.30%	2.14%	-0.13%	4.32%
		3	2.81%	-2.84%	9.71%	3.24%	-2.98%	9.38%
		4	1.54%	-1.71%	5.93%	1.89%	-1.58%	6.74%
One Thread Bias	Yes	2	-4.01%	-6.43%	-2.32%	-4.34%	-6.54%	-3.65%
		3	-1.32%	-3.18%	-0.38%	-1.34%	-3.18%	-0.37%
		4	-1.59%	-3.29%	-0.29%	-1.61%	-3.21%	-0.40%
	No	2	3.18%	1.01%	5.12%	3.12%	0.47%	5.14%
		3	4.16%	-0.53%	11.36%	4.63%	-0.61%	10.73%
		4	2.84%	0.52%	6.46%	3.08%	0.54%	6.88%
Oldest First	Yes	2	-6.13%	-10.38%	-3.07%	-6.60%	-10.16%	-3.15%
		3	-2.83%	-5.48%	-1.12%	-2.79%	-5.65%	-1.05%
		4	-2.15%	-3.85%	-1.18%	-2.16%	-3.75%	-0.85%
	No	2	1.61%	-1.30%	4.64%	1.33%	-1.13%	4.55%
		3	2.86%	-2.34%	10.33%	3.41%	-2.49%	9.59%
		4	1.76%	-1.00%	5.91%	2.01%	-0.93%	6.27%

Table 7.2: Summary of Various Selection Methods on SMT

The results clearly show that the Position Based scheme is the worst of the four techniques when the wakeup/select logic is atomic. However, the results for atomic wakeup/select logic indicate that increased support for thread contexts reduces the performance gain achieved by the other selection techniques. This is likely a result of back-to-back execution allowing additional dependents of load instructions to be issued; increasing the average replay penalty.

Increasing processor frequencies will lead to a pipelining of the scheduling logic as cycle times decrease. The Position Based scheme is simplest of the four selection policies examined and can reasonably be considered atomic. The remaining schemes involve various complexities, such as sorting and counting, which may lead them to be pipelined as cycle times decrease. Obviously, when pipelined all of the selection methods perform worse than atomic Position Based but performance losses do not exceed 5.65%.

Of the techniques examined Position Based and One Thread Bias are not recommended. The Oldest First policy works well for small amounts of TLP and had the best pipelined performance for support of 2 threaded contexts. The Distributed policy showed the best pipelined performance with more aggressive SMT designs.

Chapter 8

Conclusions and Future Work

The main contribution of this thesis is that TLP is not sufficient enough to replace the ILP extracted from speculative scheduling and speculative execution. This validates that single-threaded ILP techniques are still important components in deriving performance from SMT machines. TLP, even with an aggressive 4 threaded SMT, is not sufficient to cover up latency generated by removing the speculative techniques examined within the scope of this work. TLP is unable to cover up the pipeline bubbles that occur due to a lack of speculation, however, TLP clearly improves performance compared to the superscalar machine.

This thesis also shows that a pipelined scheduler is feasible with low performance loss. TLP is effective at reducing the penalties associated with losing back-to-back execution.

The results demonstrate that speculative execution is required to extract performance resulting in significant degradation when it is disabled. Speculative fetch was able to reduce the penalty and may warrant additional consideration in tandem with other fetching and resource allocation policies that may be able to benefit from being able to speculatively fill the instruction fetch queue when fetching would normally be denied to a particular thread. A confidence based technique may perform better with sufficient TLP

to exploit, however, it appears unlikely that avoiding speculative execution can come with any benefit considering the losses shown in Table 8.1.

	# Threads	Average IPC Performance Loss	Min. IPC Performance Loss	Max. IPC Performance Loss	Average Fairness Loss	Min. Fairness Loss	Max. Fairness Loss
DISPEX	1	64.58%	2.05%	84.56%			
	2	61.31%	37.50%	75.27%	62.23%	38.05%	75.30%
	3	54.78%	30.13%	68.21%	59.37%	31.22%	68.29%
	4	46.70%	23.44%	63.74%	46.02%	23.51%	64.15%
	8	34.55%	32.06%	38.39%	36.05%	28.82%	43.18%
DISPEX+SF	2	45.38%	15.74%	63.14%	46.55%	17.19%	63.16%
	3	35.41%	8.78%	51.66%	40.89%	10.33%	51.59%
	4	28.57%	12.76%	48.27%	29.61%	13.61%	48.94%
ADISPEX+SF	2	43.13%	14.39%	62.78%	44.00%	15.16%	62.79%
	3	33.91%	5.85%	50.80%	40.48%	8.38%	50.78%
	4	24.61%	5.36%	49.32%	25.93%	5.96%	50.33%
DYCOSPEX	2	7.64%	-22.16%	24.82%	13.02%	-3.73%	5.30%
	3	21.12%	4.96%	28.97%	26.04%	6.57%	50.27%
	4	17.19%	1.53%	39.72%	23.15%	4.12%	39.62%
COCOSPEX	2	4.75%	-24.30%	20.75%	8.83%	-10.06%	21.45%
	3	8.49%	-0.06%	17.33%	11.28%	-0.02%	17.60%
	4	2.94%	-7.74%	11.04%	2.88%	-9.74%	11.06%

Table 8.1: Summary of Disabling Speculative Execution

Load-hit prediction leaves much potential discussion. Long issue-to-execute delays increase the need for load-hit prediction but reducing this delay may not be feasible. Clearly, with reduced issue-to-execute delay performance increases and if combined with disabled speculative scheduling with sufficient TLP this can minimized performance losses (0.42% for 1 cycle issue-to-execute delay and 4 threaded SMT). It is expected that additional thread context support will result in performance improvements in this case.

Instruction replay must still be handled and the only way to truly improve the situation is to increase cache sizes or ensure that cache misses do not occur. Without speculative

scheduling the number of replays is reduced but this appears to come at the expense of overall performance shown in Table 8.2.

Disabled Load-Hit Speculation with...	# Threads	Average IPC Performance Loss	Min. IPC Performance Loss	Max. IPC Performance Loss	Average Fairness Loss	Min. Fairness Loss	Max. Fairness Loss
Issue-to-execute delay of 1 cycle	1	10.30%	-1.94%	40.28%			
	2	5.73%	-1.64%	10.71%	5.46%	-1.63%	10.55%
	3	1.80%	-0.46%	5.06%	1.68%	-0.94%	5.45%
	4	0.42%	-0.84%	1.44%	0.34%	-1.11%	1.42%
Issue-to-execute delay of 2 cycle	1	26.22%	5.24%	60.18%			
	2	20.74%	11.67%	26.15%	20.28%	11.67%	26.46%
	3	13.17%	6.11%	19.88%	13.73%	5.91%	20.81%
	4	9.61%	4.49%	19.64%	10.51%	4.62%	20.91%
Issue-to-execute delay of 3 cycle (Baseline)	1	37.90%	8.63%	70.14%			
	2	33.68%	24.76%	40.87%	32.93%	24.77%	40.22%
	3	25.50%	16.13%	33.78%	26.08%	16.17%	34.61%
	4	22.21%	15.52%	29.89%	22.64%	15.60%	31.46%

Table 8.2: Summary of Disabled Speculative Scheduling on SMT

Various 2-cycle pipelined scheduler implementations demonstrate that Oldest First performs best for 2 threaded SMT machines and Distributed performs best for 3 and 4 threaded SMT. As cycle times decrease, it will be difficult to have a complex scheme used for an atomic scheduler and pipelined schedulers will have to be considered. This work indicates that Oldest First and Distributed are potentially good methods for a pipelined scheduler but further investigation into multi-cycle scheduling logic should be considered.

Selection Method	Back-to-Back	# Threads	Average IPC Performance Loss	Min. IPC Performance Loss	Max. IPC Performance Loss	Average Fairness Loss	Min. Fairness Loss	Max. Fairness Loss
Position Based	Yes	1,2,3,4						
	No	1	9.93%	1.52%	31.88%			
		2	5.65%	2.53%	7.81%	5.63%	2.41%	7.88%
		3	4.71%	1.21%	11.95%	5.23%	1.18%	11.57%
		4	3.74%	0.78%	8.21%	3.85%	0.78%	8.17%
Distributed	Yes	2	-5.56%	-8.38%	-3.46%	-5.99%	-8.54%	-3.39%
		3	-2.96%	-6.15%	-0.38%	-3.02%	-6.25%	-0.28%
		4	-2.60%	-5.14%	-1.45%	-2.51%	-4.93%	-1.13%
	No	2	2.40%	0.08%	4.30%	2.14%	-0.13%	4.32%
		3	2.81%	-2.84%	9.71%	3.24%	-2.98%	9.38%
		4	1.54%	-1.71%	5.93%	1.89%	-1.58%	6.74%
One Thread Bias	Yes	2	-4.01%	-6.43%	-2.32%	-4.34%	-6.54%	-3.65%
		3	-1.32%	-3.18%	-0.38%	-1.34%	-3.18%	-0.37%
		4	-1.59%	-3.29%	-0.29%	-1.61%	-3.21%	-0.40%
	No	2	3.18%	1.01%	5.12%	3.12%	0.47%	5.14%
		3	4.16%	-0.53%	11.36%	4.63%	-0.61%	10.73%
		4	2.84%	0.52%	6.46%	3.08%	0.54%	6.88%
Oldest First	Yes	2	-6.13%	-10.38%	-3.07%	-6.60%	-10.16%	-3.15%
		3	-2.83%	-5.48%	-1.12%	-2.79%	-5.65%	-1.05%
		4	-2.15%	-3.85%	-1.18%	-2.16%	-3.75%	-0.85%
	No	2	1.61%	-1.30%	4.64%	1.33%	-1.13%	4.55%
		3	2.86%	-2.34%	10.33%	3.41%	-2.49%	9.59%
		4	1.76%	-1.00%	5.91%	2.01%	-0.93%	6.27%

Table 8.3: Summary of Various Selection Methods on SMT

A particularly notable concern was that the fetching policy, ICOUNT, may not be suitable for optimizing TLP with the presented techniques. ICOUNT works because it relies on issue, decode and rename queue occupancy that is well established using a baseline set of ILP exploiting techniques. However, there are problems using ICOUNT while disabling speculative execution. A fetch-stalled thread would appear to have very few instructions in the issue queue and giving it priority with ICOUNT that cannot be realized. Modifications were made to ICOUNT to overcome this problem but were not fully

investigated as it is not expected to improve performance enough to justify disabling speculative execution. It is important to realize that properties of ICOUNT may not be conducive to future work and to consider various different fetching policies.

References

[BA 97] Berger, D., Austin, T. "The SimpleScalar tool set: Version 2.0", Technical Report, Department of Computer Science , University of Wisconsin-Madison, June 1997.

[BSP 01] Brown, M., Stark, J., Patt, Y. "Select-Free Instruction Scheduling Logic", 34th International Symposium on Microarchitecture, December 2001.

[BP 92] Butler, M., Patt, Y. "An Investigation of the Performance of Various Dynamic Scheduling Techniques", 25th International Symposium on Microarchitecture, 1992, p.1-9.

[CE 98] Chrysos, G.Z., Emer, J.S. "Memory Dependence Prediction using Store Sets", 25th International Symposium on Computer Architecture, June 1998.

[CF+ 03] Cazorla, F.J., Fernandez, E., Ramírez, A., Valero, M. "Improving Memory Latency Aware Fetch Policies for SMT Processors", 5th International Symposium on High-Performance Computing, 2003, p. 70-85.

[CR 00] Calder, B., Reinman, G. "A Comparative Survey of Load Speculation Architectures", Journal of Instruction-Level Parallelism, May 2000.

[CR+ 04] Cazorla, F.J., Ramírez, A., Valero, M., Fernandez, E. "Dynamically Controlled Resource Allocation in SMT Processors", 37th International Symposium on Microarchitecture, 2004.

[CY 06] Choi, S., Yeung, D. "Learning-Based SMT Processor Resource Distribution via Hill-Climbing", 33rd International Symposium on Computer Architecture, 2006.

[EA 03] El-Moursy, A., Albonesi, D.H. "Front-End Policies for Improved Issue Efficiency in SMT Processors", 9th International Symposium on High-Performance Computer Architecture, February 2003.

[EP+ 98] Evers, M., Patel, S.J., Chappell, R.S., Patt, Y.N. "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work", 25th International Symposium on Computer Architecture, 1998, p. 52-61.

[FR+ 02] Martínez, J.F., Renau, J., Huang, M.C., Prvulovic, M., Torrellas, J. "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors", 35th International Symposium on Microarchitecture, 2002.

[GA+ 05] Gandhi, A., Akkary, H., Rajwar, R., Srinivasan, S.T., Lai, K. "Scalable Load

and Store Processing in Latency Tolerant Processors", 32nd International Symposium on Computer Architecture, 2005.

[H 00] Henning, J.L. "SPEC CPU2000: Measuring CPU Performance in the New Millennium", IEEE Computer, July 2000, p. 28-35.

[HP 97] Hennessy, J.L., Patterson, D.A. "Computer Organization and Design", 2nd Edition, 1997.

[J 03] Jiménez, D.A. "Reconsidering Complex Branch Predictors", 9th International Symposium on High Performance Computer Architecture, 2003.

[KL 04] Kim, Ilhyun., Lipasti, M.H. "Understanding Scheduling Replay Schemes", 10th International Symposium on High Performance Computer Architecture, 2004, p. 198.

[LGF 01] Luo, K., Gummaraju, J., Franklin, M. "Balancing Throughput and Fairness in SMT Processors", International Symposium on Performance Analysis of Systems and Software, January 2001, p. 161-171.

[LWS 96] Lipasti, M.H., Wilkerson, C.B., Shen, J.P. "Value Locality and Load Value Prediction", Architectural Support for Programming Languages and Operating Systems, 1996.

[M 93] McFarling, S. "Combining Branch Predictors", DEC Western Research Laboratory Technical Note TN-36, June 1993.

[MB 02] Moreshet, T., Bahar, R.I. "Complexity-Effective Issue Queue Design Under Load-Hit Speculation", Workshop on Complexity-Effective Design, May 2002.

[MLO 01] Morancho, E., Llabería J.M., Olivé, À. "Recovery Mechanism for Latency Misprediction", 2001 International Conference on Parallel Architectures and Compilation Techniques, 2001.

[MKP 06] Mutlu, O., Kim, H., Patt, Y.N. "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance", IEEE Micro 26, 2006, p. 10-20.

[ON+ 96] Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K., Chang, K. "The Case for a Single-Chip Multiprocessor", 7th International Symposium Architectural Support for Programming Languages and Operating Systems, October 1996.

[PJS 97] Palacharla, S., Jouppi, N.P., Smith, J.E. "Complexity-Effective Superscalar Processors", 24th International Symposium on Computer Architecture, 1997, p. 206-218.

[PSR 92] Pan, S., So, K., Rahmeh, J.T. "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", 5th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, p. 76- 84.

[RFL 03] Ramsay, M., Feucht, C., Lipasti, M.H. "Exploring Efficient SMT Branch Predictor Design", ISCA Workshop on Complexity Design, June 2003.

[S 81] Smith, J.E. "A Study of Branch Prediction Strategies", Proceedings of ISCA-8, 1981.

[SBP 00] Stark, J., Brown, M.D., Patt, Y.N. "On Pipelining Dynamic Instruction Scheduling Logic", 33rd International Symposium on Microarchitecture, 2000.

[SPG 05] Sharkey, J.J., Ponomarev, D., Ghose, K. "M-SIM: A Flexible, Multithreaded Architectural Simulation Environment", Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton, October 2005.

[TB 01] Tullsen, D.M., Brown, J.A. "Handling Long-Latency Loads in a Simultaneous Multithreading Processors", 34th International Symposium on Microarchitecture, December 2001.

[TE+ 96] Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", 23rd Annual Symposium on Computer Architecture, May 1996.

[TEL 95] Tullsen, D.M., Eggers, S., Levy, H. "Simultaneous Multithreading: Maximizing On-Chip Parallelism", 22nd Annual International Symposium on Computer Architecture, June 2005.

[TT 03] Tuck, N., Tullsen, D.M. "Initial Observations of the Simultaneous Multithreading Pentium 4 Processor", 12th International Conference on Parallel Architectures and Compilation Techniques, September 2003.

[YE+ 99] Yoaz, A., Erez, Mattan., Ronen, R., Jourdan, S. "Speculation Techniques for Improving Load Related Instruction Scheduling", 26th International Symposium on Computer Architecture, May 1999.