2004

An evaluation of multiple branch predictor and trace cache advanced fetch unit designs for dynamically scheduled superscalar processors

Slade S. Maurer
Louisiana State University and Agricultural and Mechanical College, slade@computer.org

AN EVALUATION OF MULTIPLE BRANCH PREDICTOR AND TRACE CACHE ADVANCED FETCH UNIT DESIGNS FOR DYNAMICALLY SCHEDULED SUPERSCALAR PROCESSORS

A Thesis
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in
The Department of Electrical and Computer Engineering

by
Slade S. Maurer
B.S., Southeastern Louisiana University, Hammond, Louisiana, 1998
December 2004

ACKNOWLEDGEMENTS

This thesis would not be possible without several contributors. I want to thank my advisor Professor David Koppelman for his guidance and support. He brought to my attention this fascinating thesis topic and trained me to modify RSIML to support my research. I would also like to thank Professor Jagannathan Ramanujam for his rich classroom lectures and inspirational work ethic. I want to thank my family for their support and encouragement. This thesis is dedicated to my wife Simone Flory whose love, intelligence and experience have been invaluable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Semiconductor feature size continues to decrease permitting superscalar microprocessors to continue to increase the number of functional units available for execution. As the instruction issue width increases beyond the five instruction average basic block size of integer programs, more than one basic block must be issued per cycle to continue to increase instructions per cycle (IPC) performance. Researchers have created methods of fetching instructions beyond the first taken branch to overcome the bottleneck created by the limitations of conventional single branch predictors.

We compare the performance of the multiple branch prediction (MBP) and trace cache (TC) fetch unit optimization methods. Multiple branch predictor fetch unit designs issue multiple basic blocks per cycle using a branch address cache and a multiple branch predictor. A trace cache uses the runtime instruction stream to create fixed length instruction traces that encapsulate multiple basic blocks.

The comparison is performed by using a SPARC v8 timing based simulator. We simulate both advanced fetch methods and execute benchmarks from the SPEC CPU2000 suite. The results of the simulations are compared and a detailed analysis of both microarchitectures is performed.

We find that both fetch unit designs provide a competitive IPC performance. As issue width is increased from an eight to sixteen way superscalar, the IPC performance improves implying that these fetch unit designs are able to take advantage of the wider issue widths.

The MBP can use a smaller L1 instruction cache size than the TC and yet achieve a similar IPC performance. Pre-arranged instructions provided by the TC allow the

pipeline stages to be shortened in comparison to the MBP. The shorter pipeline significantly improves the IPC performance.

Prior trace cache research used two or more ports to the instruction cache to improve the chances of fetching a full basic block per cycle. This was at the expense of instruction cache line realignment complexity. Our results show good performance with a single instruction cache port.

We study an approximately equal cost implementation for the MBP and TC. Of the six benchmarks studied, the TC outperforms the MBP over four of the benchmarks.

CHAPTER 1. INTRODUCTION

As semiconductor feature size decreases the number of functional units included on a microprocessor die increases. To accommodate this trend researchers must overcome several architectural hurdles and one significant area of research focuses on improving the instruction fetch rate.

Most current superscalar microprocessors utilize single branch prediction to improve the peak fetch rate performance up to the basic block boundary of four to six instructions for integer programs for non-contiguous basic blocks. The fetch rate is improved such that the issue rate can be improved to be able to issue several instructions per cycle to multiple functional units.

The basic block is integral to the understanding of the proposed architectures so here we will define it. We define a *basic block* to be a sequence of instructions that contain a single control transfer instruction (CTI) at the end (there may be a delay slot following the CTI). The instructions in a basic block have sequential program counter values. The basic block has only one exit point and might have more than one entry point. As a side note, a *delay* slot is an instruction that immediately follows a branch and is executed before the branch target. Our definition of a basic block is slightly looser than the accepted definition since we permit more than one entry point for the reason that it is more useful for describing the design of our MBP. The accepted definition of a basic block is as follows, "*basic block* – a straight-line code sequence with no branches in except to the entry and no branches out except at the exit" [HP03, pp. 173].

Researchers have suggested advanced fetch unit architectures that are capable of issuing multiple basic blocks per cycle. This thesis will focus on two advanced fetch unit

architectures that have been suggested in the literature and which demonstrate significant fetch rate performance improvements over conventional single branch predictor fetch unit architectures [YMP93, RBS96].

The first architecture we will examine is referred to as a multiple branch prediction fetch unit architecture. *Multiple branch prediction* fetch unit architectures are composed of a multiple branch predictor, branch address cache, multi-ported instruction cache, non-trivial realignment logic, and may contain other small internal buffers. This optimization technique uses the multiple branch predictor and branch address cache to speculate the directions and targets of several branches and then fetch the instructions from the multi-ported instruction cache in a single cycle. Research has demonstrated significant performance improvements through the use of this design [YMP93].

The second architecture under study is the trace cache fetch unit architecture. *Trace cache* based designs contain a trace cache, next trace predictor, single port instruction cache, simple realignment logic, and may contain other small internal buffers. This optimization technique leverages atomic *traces* of instructions which are a fixed length, encapsulating multiple basic blocks, dynamically created at runtime from instructions that are fetched from the single port instruction cache. Research has demonstrated significant performance improvements through the use of this design [RBS96].

Little research exists which directly compares the performance and complexity of the multiple branch prediction and the trace cache fetch unit architectures. Additionally, the simulations upon which the existing research are based are not full timing based dynamically scheduled superscalar simulations using an industry standard RISC

instruction set. The researchers who studied each of the architectures used different simulator designs, simulation benchmarks, instruction sets and simulated architectures.

The goal of this thesis is to provide a detailed and rigorous comparison of these two microarchitectural techniques on a full timing based superscalar microprocessor using an industry standard instruction set. The simulated microprocessor is based on a real-world instruction set. We use SPEC benchmarks for simulation and identical hardware configurations such as L1 instruction cache size, issue width and reorder buffer depth.

Comparing the two fetch unit architectures is interesting for a few reasons. The two designs are significantly different but accomplish similar goals so a direct comparison will provide insight into the design tradeoffs. An important tradeoff to be examined is a direct comparison of implementation size versus performance for the fetch unit designs under study, instruction cache complexity, and the "sweet spot" of the size versus performance for the various structures used in the fetch units. Additionally, we study the effect of shortening the pipeline length of the trace cache design. Varying the fetch pipeline is studied because the realignment logic may be less complex than that present in the multiple branch predictor [RBS96].

CHAPTER 2. BACKGROUND

In this chapter we present background information useful for the understanding of the two simulated fetch unit architectures.

2.1 Single Branch Prediction

Dynamic branch prediction is an architectural feature designed to determine the direction of a branch at runtime, before the direction is resolved through the execution of the branch instruction. The simplest dynamic branch prediction schemes produce one direction prediction per cycle. The target of the predicted branch may not be determined until the decode stage, the earliest pipeline stage when a branch target can be resolved, unless a branch target buffer or a similar technique is used. The fetch unit uses the direction prediction to determine the next basic block of instructions to fetch. In the case of a taken branch, fetching begins once the branch target can be determined. *Single branch prediction* is a dynamic branch prediction technique that uses a branch-prediction buffer or a branch history table to determine the direction of a single branch in a single cycle.

J. Smith covers many branch direction prediction schemes including the bimodal predictor [S81]. Lee et al. evaluate several branch direction predictor designs and introduce branch target buffers to reduce pipeline delay by determining the branch target in the same cycle as the prediction [LS84]. McFarling et al. compare hardware and software approaches including profiling [MH86]. Pan et al. describe combining global and local history into a dynamic branch predictor [PSR92]. Yeh et at. introduce local and global branch direction prediction schemes [YP92, YP93]. S. Mcfarling decomposes early single branch predictor architectures into subproblems and studies these in detail;

which produces the GSHARE designs (Global History with Index Sharing) [M93]. The YAGS branch predictor is proposed by Eden et al. as a high performance single branch predictor design [EM98].

2.2 Target Buffer

A *target buffer* is a PC indexed table that returns the target of a control transfer instruction at that program counter. It stores the target of a control transfer instruction for later use by the fetch unit to quickly determine a target before it can be resolved by stages further in the pipeline. This is a generalization of the *branch target buffer* as defined by Hennessy et al. [HP03, pp. 209].

2.3 Multiple Branch Predictor

Multiple branch prediction extends single branch prediction using a multiple branch predictor. We will use the acronym *MBP* for the **M**ultiple **B**ranch **P**redictor.

A *multiple branch predictor* is a microarchitectural design that simultaneously looks-up multiple branch direction predictions in a single cycle. A *branch address cache* is used by the MBP to simultaneously determine the branch targets of each predicted branch. The MBP requires a multi-ported instruction cache that can issue more than one line per cycle. This is done to permit the MBP fetch unit design to be able to fetch multiple basic blocks by using branch direction predictions and target addresses.

Realignment logic is required to format the noncontiguous basic blocks contained in the lines fetched from the instruction cache into a contiguous block of instructions to be passed to the decode latch.

A conventional single branch predictor can be used to predict multiple branch directions by using chained-lookups. This technique requires each lookup to be

performed in series – therefore increasing the cycle time. This simple method is wasteful of cycle time prompting researchers to develop innovative solutions that can perform more than one prediction simultaneously. Multiple branch predictors are more sophisticated than single branch predictors performing chained-lookup, they attempt to predict multiple branch outcomes in a single cycle while minimizing storage requirements.

T. Yeh et al. propose the multiple branch predictor and branch address cache [YM93]. The multiple branch predictor design of Yeh et al. uses shifted versions of the global history register to predict more than one branch direction per cycle. A collapsing buffer design is introduced to shorten the decode latch realignment stages of the pipeline required by the multiple branch predictor by Conte et al. [CMMP95]. Dutta et al. research the tree-like subgraph that is very much like a multiple branch predictor with extensive use of local history [DF95]. Seznec et al. propose the pipelined and cost-effective two-block ahead predictor that uses the current block to predict the next blocks [SJSM96]. Wallace et al. simulate a MBP design similar to Yeh et al., but use two bit counters for each embedded block instruction and a scalable look-up technique [WB97]. D. Koppelman analyzes a multiple branch predictor design using a YAGS based predictor and specialized branch address cache which include indirect jump prediction on a full timing based simulator [K02].

2.4 Trace Cache

The trace cache fetch unit design is motivated by the early MBP designs proposed in the literature. Rotenberg et al. speak to the motivation of their proposed TC design in the section “Problems with other Fetch Unit Mechanisms”:

> "Recall that the job of the fetch unit is to feed the dynamic instruction stream to the decoder. Unlike the trace cache approach, previous designs have only the conventional instruction cache, containing a static form of the program, to work with. Every cycle, instructions from non-contiguous locations must be fetched from the instruction cache and assembled into the predicted dynamic sequence. […] The trace cache approach avoids these problems by caching dynamic instruction sequences themselves, ready for the decoder" [RBS96, pp. 5].

The *trace cache* fetch unit design creates a special instruction cache internal to the fetch unit, the *trace cache*, which is used to hold instructions in their dynamic runtime order. A *trace* is a snapshot of the dynamic instruction stream that is at most *n* instructions in length and is composed of instructions in their runtime order. The trace is at most *n* instructions long so that it can be cached in the trace cache that has a line length of *n* instructions. We will use the acronym *TC* for the **T**race **C**ache fetch unit design.

The trace cache design has two conceptual levels of sequencing: instruction level and trace level. Instruction level sequencing, we will use the acronym *ILS*, is the initial state of the fetch unit and performs its work with the instruction as the atom of operation. Trace level sequencing, we will use the acronym *TLS*, is entered from ILS and performs its work using the trace as the atom of operation.

Melvin et al. originate the central concept of the trace cache, which is large atomic units of work composed of multiple instructions, and introduce the fill unit [MSP88]. An early trace cache design is introduced based on multiple branch prediction by Rotenberg et al. [RBS96]. A trace processor design was developed to explore a new architecture built around the trace concept [RJSS97]. An improved trace cache design using the next trace predictor was studied by Rotenberg et al. [RBS99].

CHAPTER 3. INSTRUCTION SET AND FETCH UNIT ARCHITECTURES

In this chapter we discuss the instruction set architecture used for simulation. Then we cover the multiple branch prediction fetch unit design implemented in our simulations. Finally, we detail the simulated trace cache fetch unit design.

3.1 SPARC V8

SPARC is an acronym for **S**calable **P**rocessor **ARC**hitecture. It was first announced in 1987 by SPARC International. It follows the business model that a single company controls the design of the instruction-set architecture and then licenses the right to create a specific implementation. For this reason, several companies sell microprocessors which are their proprietary implementation of the SPARC instruction-set architecture specification.

SPARC architectures are RISC based and incorporate many of the features characteristic of this design methodology. It is a load-store architecture, achieve efficient pipeline implementation, fixed length instructions, and most instructions can execute in a single cycle.

The SPARC instruction set is guaranteed to be backwards compatible by SPARC International. It can used in many applications from low cost embedded devices all the up to high performance supercomputers. Backward compatibility at the binary level is key to the industry acceptance of most processor architectures because of the large investment customers have in software compiled for a specific instruction set.

SPARC V8 architecture is designed to incorporate many of the microarchitectural advances of the 1980's and early 1990's. It incorporates integer data, selectable big and

little endian byte orders, more registers, greater support for optimizing compilers, and superscalar design.

The SPARC V8 instruction set and processor design is used within this thesis. It is a good choice for the full qualification and comparison of the fetch unit microarchitectural techniques discussed herein because it is a typical and widely used RISC ISA.

The choice of a RISC ISA is important. RISC ISAs are among the fastest and CISC ISAs have become outdated. In fact, some of the newer CISC x86 implementations use translation to map the externally input CISC instructions to one or more internally executed RISC micro-operations. VLIW ISAs are more complex, are less studied than RISC, and have unproven real-world implementations. For these reasons VLIW ISAs are not a good choice for our simulations.

Other reasons that the SPARC v8 ISA is chosen for simulation are that it has easy to decode RISC instructions, explicit superscalar support and a robust design.

3.2 Multiple Branch Predictor

This section will focus on the description of the MBP design used for simulation experiments in this thesis. More information on other MBP designs can be found in Chapter 2 of this document.

The multiple branch prediction fetch unit architecture used in our simulations is a tree based architecture simulated by Koppelman [K02] and designed by Yeh et al. [YMP93]. It has two major microarchitectural components: the branch address cache and the multiple branch predictor.

3.2.1 Branch Address Cache

The *branch address cache*, in a single cycle, uses the program counter to determine a specific tree of basic blocks to a certain height that follow the PC in the dynamic instruction stream. Koppelman describes the BAC design as follows, "A PC-indexed branch address cache (BAC) provides information on the tree of basic blocks within a number, say 3, of the program counter (PC) …" [K02].

The basic block tree is used by the MBP, in conjunction with the direction prediction of each branch simultaneously determined by a multiple branch predictor, to select a path through the basic block tree. The complete path information is used to fetch multiple basic blocks in a single cycle from a multi-ported instruction cache.

The BAC is critical to the performance of the MBP architecture. It permits the retrieval of multiple branch targets in a single cycle. The PC indexed BAC contains a tree of all the basic blocks reachable from a PC to a certain distance [YMP93, K02]. A tree node contains all the information required to reconstruct the path through the tree including the length of the basic block, the type of control transfer instruction at its end, and potentially the CTI's target.

3.2.2 YAGS Multiple Branch Predictor

The MBP uses a specialized multiple branch predictor that is able to predict multiple branch outcomes in a single cycle. It is designed such that it can predict the number of outcomes required to determine the path through a tree of basic blocks contained in the BAC. The height of the BAC tree is defined as the *order* of the BAC [K02]. As such, the order of the BAC is equal to the number of predictions the multiple predictor must make in a single cycle to specify a full path through a BAC path tree.

In one cycle, the multiple branch predictor is used to determine multiple direction predictions to a certain order using the first branch PC. Simultaneously, the BAC is indexed using the first branch PC to retrieve a basic block tree of the same order. Then, the direction predictions from the multiple branch predictor are used to determine a specific path through the selected BAC basic block tree. Finally, the predicted path is used to fetch multiple basic blocks from the multi-ported instruction cache.

Using a single branch predictor and chaining the lookups to produce multiple predictions must be avoided because this would violate the single cycle requirement and would not scale well. The MBP does not use local path history because only the address of the first basic block is known at lookup since lookups occur across multiple basic blocks per cycle and lookups are not chained. This point rules out local history strategies.

The branch prediction scheme used in the MBP simulations is based on the YAGS predictor. *YAGS* is an acronym for **Y**et **A**nother **G**lobal **S**cheme and, as the name implies, is a global history based predictor. It contains a primary branch history table, we will use the acronym *PBHT*, and two smaller tables that are used to reduce the amount of aliasing. The PBHT uses a two bit counter for prediction, stores the branch history and is indexed only using the branch PC.

The secondary history tables, we will use the acronym *SHT*, each have entries that have a partial PC used as a tag to identify aliasing and are indexed using the PC XORed with a global history register. When the tag matches then the SHTs are used, otherwise the PBHT is used. The particular SHT chosen for prediction is determined by the counter in the PBHT entry. If the PBHT entry predicts a taken branch then the “not-taken” SHT is checked for a matching tag and if the PBHT predicts not-taken then the “taken” SHT is referenced.

The index used to lookup a prediction within the table is created by XORing the global history shift register and the branch PC. YAGS has demonstrated superior performance across many benchmarks when compared to other global schemes with equivalent chip area [EM98]. We will use the acronym *GHR* for the Global History Register.

The YAGS predictor has been modified to support multiple branch predictions by creating multiple prediction ports from the predictor. Each prediction is performed on the index generated by the YAGS prediction scheme. The index is computed from the XOR of the global history register and the first branch PC at the root of the BAC tree being referenced. The entire GHR is used to make the first basic block's prediction, a single bit shifted GHR is used for the next block, and subsequent predictions are determined from p-1 shifted versions of the GHR XORed with the root branch PC [K02]. The technique of shifting the GHR to produce multiple direction predictions simultaneously is similar to and based on the technique used in Yeh et al.'s MBP [YMP93].

The MBP is updated once a branch's direction has resolved. If an entry in a SHT predicted the resolving branch's direction then the processor uses state information to update its counter. The PBHT entry's counter is updated unless its prediction contradicts a SHT entry that provided the prediction for the resolving branch.

3.2.3 BAC/MBP Update

The BAC entries and MBP entries are created as branches are decoded. During decode stage the branch target is determined. This information is used to populate the BAC. A BAC entry is built incrementally when an entry is referenced and its node tree is not as deep as the order of the BAC. In this case the BAC enters fill mode. When the

branch at the bottom of the entries tree is decoded its target information and other data is updated thereby extending the BAC entry's tree.

3.2.4 Multi-Ported Instruction Cache

The multi-ported instruction cache, which is required for the operation of the MBP, is designed to provide the non-contiguous access of multiple basic blocks in a single cycle. The multiple basic blocks needed in a cycle by the MBP may or may not be contiguous depending on the direction predictions of the multiple branch predictor. Even in the case that all of the branch direction predictions are not-taken, a path of basic blocks may cross an instruction cache line boundary thereby requiring more than a single cache line to be fetched in a cycle.

3.2.5 Decode Latch Realignment

Once instructions have been fetched from the multi-ported instruction cache, they must be aligned so that they can be latched into the decode stage. *Realignment logic* is used to align instructions that are fetched from instruction cache lines in their compiled static order into their dynamic runtime order. This realignment logic is non-trivial and is itself an area of research. According to Rotenberg et al., the pipeline latency of the realignment logic will be at least three logic layers when using the multi-ported instruction cache output directly:

> "Three stages of logic are required after the base instruction cache. It will be shown that this design adds much less delay than other cache designs which attempt to align non-contiguous blocks of instructions in the fetch path. […] The three functions that are performed are (1) masking off unused instructions, (2) possibly interchanging the output of the two cache banks, (3) left shifting the instructions into the final instruction latch by an arbitrary amount." [RBS96, pp. 40].

One proposed method of further reducing the realignment logic and its latency is the collapsing buffer [CMMP95] however this design is not simulated in our work.

Our MBP simulation does not simulate a specific realignment design and simply assumes a fixed number of stages in the fetch stage pipeline to accommodate the realignment of instructions from the multi-ported cache. Instructions traveling through the fetch stage pipeline are delayed the specified number of cycles. Please refer to the section regarding Latency and Timing of Pipelines in Chapter 4 for further details.

3.3 Trace Cache

This section will focus on the description of the trace cache fetch unit design used for simulation experiments in this thesis. More information on other trace cache designs can be found in Chapter 2 of this document.

The TC fetch unit design is used to fetch a single basic block per cycle in its compile-time order from a singly ported instruction cache during ILS or multiple basic blocks per cycle in their run-time order from the trace cache during TLS.

ILS is designed for simplicity. It can fetch up to all of the instructions in a single instruction cache line with the restriction that all the instructions fetched in a cycle are from a single basic block. No attempt is made to fetch instructions from a target basic block after the first block is fetched from a cache line even if the target basic block's instructions are contained in the same cache line.

3.3.1 Novel Features

Novel features are designed into our trace cache fetch unit design. We have modified the Jacobson et al. next trace predictor and Rotenberg et al. trace cache indexing to support the inclusion of indirect jump target basic blocks in a trace. Our trace selection algorithm is designed to allow the fetch instruction to be embedded in the trace – not just

the first instruction. These innovations are designed to minimize the size of the trace cache by reducing duplication and fragmentation.

3.3.2 Instruction Cache

Instructions initially enter the fetch unit from the instruction cache. A simple design is used to fetch up to one basic block from the singly ported instruction cache during ILS in a cycle. Conceptually, ILS works like a conventional fetch unit in that it fetches up to a single basic block per cycle from the instruction cache.

Other TC design research has used an interleaved L1 instruction cache so that two or more consecutive lines can be fetched in a single cycle. Researchers using a multi-ported instruction cache design for a trace cache implementation include: Rotenberg et al. [RBS96] and [RBS99], Postiff et al. [PTM98], and Vandierendonck [VRBV02]. Rotenberg et al. speak to the reasoning behind a multi-ported IC, “this allows fetching sequential code that spans a cache line boundary, always guaranteeing a full cache line or up to the first taken branch”.

We use a singly ported L1 instruction cache because we want to simplify the realignment logic required for ILS. Our design allows for a simple and low latency algorithm for aligning instructions. The following two logic layers illustrate what is performed during ILS to align the fetched instructions:

1. left-shifter to align the instructions into the instruction latch equal to the issue width
2. masking logic to eliminate the unused instructions

Our singly-ported L1 instruction cache design minimizes the stages in the fetch pipeline and justifies a lower latency fetch unit in comparison to the multi-ported L1

instruction cache based MBP design. Our cache design should allow fewer pipeline stages than prior TC research that required more than one port.

3.3.3 Target Buffer

A target buffer is used to provide CTI target addresses to the fetch unit before they are determined during the decode stage. Our target buffer is a direct mapped cache that masks the CTI PC's effective bits to generate the index. An entry in the target buffer is composed of the target PC and a tag. In the case of indirect jumps, the target is speculative and may trigger a target misprediction when the actual target is resolved during execution. The target buffer holds conditional/unconditional branch, direct/indirect jump, call and return targets.

3.3.4 ILS Transition Conditions

Fetching during ILS stops after any CTI is encountered or when the end of an instruction cache line is reached. The fetch unit will continue fetching instructions the next cycle if the target buffer contains the target of the CTI. If the target cannot be determined, then the fetch unit will wait until the CTI target has been resolved during decode stage. If the CTI is a branch instruction then the single branch predictor is consulted for a direction prediction. In the case of a not taken branch, the target PC is not required and fetching will begin the next cycle.

3.3.5 Branch Predictor

The branch predictor is a simple bimodal design. It uses the PC of the branch being predicted to index a branch history table. We will use the acronym *BHT* for the branch history table. Masking the effective address bits of the branch PC creates the BHT

index. Each BHT entry is composed of a 2 bit saturating counter from which the prediction is made. The counter is updated when the outcome of the branch is determined after execution.

3.3.6 CTI Misprediction

The fetch unit handles misprediction of indirect jumps and conditional branches after they are detected. Upon misprediction, the correct path is fetched and prediction hardware is updated to account for the misprediction.

3.3.7 TLS Conditions

Trace level sequencing is attempted every cycle. The only time trace level sequencing is not attempted is during the minority of cycles in which an unresolved ILS indirect jump redirection is in process or the processor is in the privileged state. An unresolved ILS jump redirection occurs when an indirect jump instruction has executed and the target PC is being fetched but has not yet been issued during ILS and the target could not be found in the target buffer.

3.3.8 Trace Cache Index

A new trace index is generated to insert traces into the trace cache each time a new trace is created. The trace cache index is created using a new algorithm tailored for the unique features of our design. Rotenberg et al. use a trace cache index generation algorithm based on their trace ID that is not suitable for our design. Our trace cache includes more CTI types such as indirect jumps and has a different trace selection design (these topics are covered in detail later in this section) so a new trace cache index generation algorithm is required.

We will provide a brief summary of the Rotenberg et al. trace cache index generation and trace selection algorithms. The index generation algorithm creates a trace cache index by masking the first PC in a trace. The first PC is called the *starting address*. The starting address is stored in a trace ID along with the branch outcomes in the trace when the trace cache is updated. The newly created trace ID is stored in the next trace predictor. A trace ID is provided by the next trace predictor when it predicts a trace to fetch from the trace cache. Trace selection begins by masking a trace ID's starting address to index the trace cache. Next, the trace is passed to the decode latch if the starting address matches the trace's tag and the trace ID's outcomes match the branch outcomes in the trace [RBS99].

The trace cache indexing technique employed by Rotenberg et al. does not fit well with the inclusion of indirect jump target basic blocks and a direct mapped trace cache. Rotenberg et al. perform trace cache indexing by masking the starting address in a trace. This implies that a direct mapped trace cache will have aliasing pressure from traces that are added to the trace cache overwriting prior traces that have different control flow but the same starting address. Rotenberg et al. do not include the target basic block of an indirect jump in a trace but our design does include this category of target CTI basic blocks. By including indirect jump target basic blocks in a trace, the branch outcomes stored in the trace ID no longer are enough to uniquely identify a trace.

Figure 1 shows the index generation algorithm. The trace cache index is generated by using a XOR hash of shifted versions of the effective 30 bits of each instruction PC contained in a trace cache line. Each instruction is shifted by a number of bits to the left equal to its position in the trace and then XORed with the other instructions in the trace. The result of the XORed instructions is then XORed with the first 5 outcomes of the

conditional branches embedded in the trace with the low order bit equal to the first branch outcome and increasing from there. The result is masked to create the trace cache index. This index is used to insert the new trace into the trace cache. It is also is used to update the next trace predictor.

Our trace cache index generator is sensitive to the instructions in the trace and their position in the trace as well as the outcomes of the branches in the trace. The left shifting is performed to track the position of each instruction in the trace and to take advantage of the lower PC bits that tend to change more than the higher bits. The generator does not require information regarding instruction type simplifying its design and increasing the scope of trace termination policies it can be applied to.

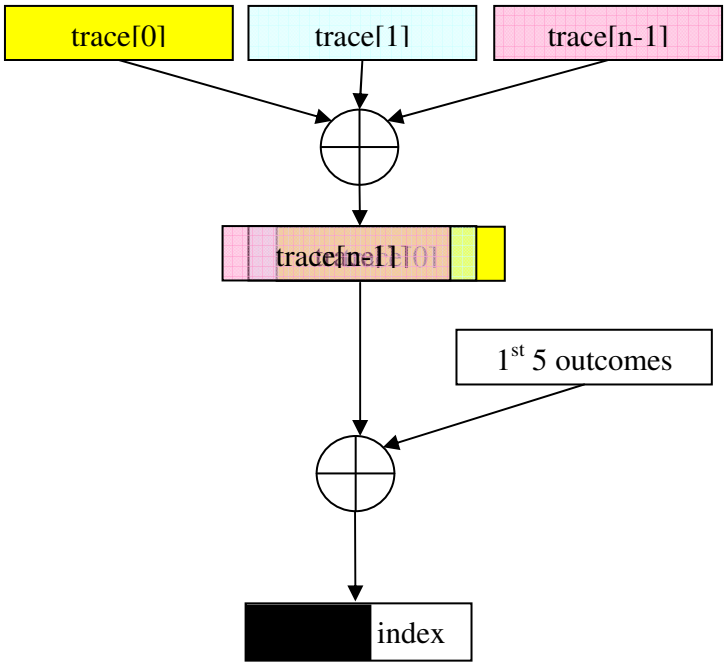

Figure 1. Our Trace Cache Index Generation Algorithm

3.3.9 Trace Selection (TSB Algorithm)

The process of *trace selection* is used to determine if a trace, which is present in the trace cache and predicted by the NTP, can be passed to the decode latch. In our design, TLS uses the fetch PC that must be fetched to compare with the selected trace from the trace cache. The fetch PC can be determined from the processor core.

Our design incorporates a novel approach to trace selection that is designed to reduce basic block duplication in the trace cache, improve the learning time of new loops and improve the correct path hit rate. This is accomplished using the *trace search back* algorithm.

When using trace search back trace selection, each instruction in the trace is compared from the beginning to the end of the trace with the fetch PC. If the trace contains the fetch PC then all the instructions from that instruction to the end of the trace are fetched.

Our trace selection technique differs from that used by Rotenberg et al. Their technique is as follows:

> "The fetch address is used together with the multiple branch predictions to determine if the trace read from the trace cache matches the predicted sequence of basic blocks. Specifically, a trace cache hit requires that (1) the fetch address matches the tag and (2) the branch predictions match the branch flags. The branch mask ensures the correct number of bits are used in the comparison." [RBS96].

Trace search back trace selection can be implemented in hardware by using a bank of parallel comparators that compare the required PC to each instruction in the candidate trace. This is simply an extension of the comparator used in the Rotenberg et al. trace selection algorithm that compares the fetch PC to the first PC in the candidate trace.

It is best to use an example to illustrate the effectiveness of our trace selection algorithm. Figure 2 provides an example of a simple loop. The loop encompasses three basic blocks: A, B and C. Let's assume that A ends with a conditional branch that is taken for several iterations. B ends in an indirect jump to C that in turn ends in an indirect jump back to A. Basic block A is five instructions in length, B is nine and C is four. For our example, we will assume the trace cache contains traces of eight instructions in length.

The sum of the instructions in the example given is 18 instructions. Let's say that the first instruction of basic block A is the first instruction of a trace and the NTP picks the correct trace to fetch from the trace cache. Also, the NTP chooses the correct path through the trace cache to fetch several iterations of the example loop. Finally, the trace termination policy allows the inclusion of indirect jump target basic blocks in a trace.

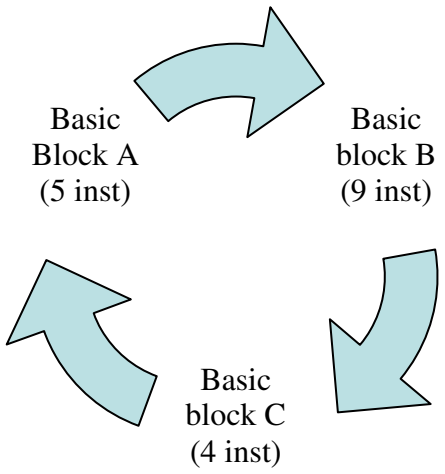

Figure 2. Simple Loop Example

Without trace search back, or *TSB*, several traces would need to be selected to fetch the instructions in this loop. This is due to the starting address of a trace being used

as the first fetched PC. This inflexible approach forces duplication in the trace cache when loops are traversed.

Given below is a list of the minimum traces required to complete a circular path through the trace cache of the example loop without TSB and with the stated assumptions. By a circular path, we mean that the path of fetched instructions completes a circuit in the trace cache by beginning with a trace and continuing until that trace is again selected.

[A0, A1, A2, A3, A4, B0, B1, B2] => [B3, B4, B5, B6, B7, B8, C0, C1] =>

[C2, C3, A0, A1, A2, A3, A4, B0] => [B1, B2, B3, B4, B5, B6, B7, B8] =>

[C0, C1, C2, C3, A0, A1, A2, A3] => [A4, B0, B1, B2, B3, B4, B5, B6] =>

[B7, B8, C0, C1, C2, C3, A0, A1] => [A2, A3, A4, B0, B1, B2, B3, B4] =>

[B5, B6, B7, B8, C0, C1, C2, C3] => **[A0, A1, A2, A3, A4, B0, B1, B2]**

It is clear from the list above that it requires nine unique traces to handle the example loop if the first PC of a trace always begins the TLS fetch.

TSB gives the fetch unit the flexibility to fetch instructions from any starting point within the trace. The following list shows how this capability reduces the number of traces required to traverse the example loop.

[A0, A1, A2, A3, A4, B0, B1, B2] => [B3, B4, B5, B6, B7, B8, C0, C1] =>

[C2, C3, A0, A1, A2, A3, A4, B0] => **[~~A0, A1, A2, A3, A4, B0,~~ B1, B2]**

The list above demonstrates the reduction of duplication in the trace cache as a result of using trace search back. The amount of traces required to represent the example loop in the trace cache has been reduced from nine to three – saving 66% of the space!

Many trace cache designs will attempt to terminate a trace at a control transfer instruction in an attempt to minimize duplication, which often results in not completely

filling a trace. That may minimize duplication, but in the example given above it would result in no more than one block per cycle being fetched. Also, the technique results in increased fragmentation in the trace cache. Duplication and fragmentation are defined in the following sections.

TSB is intended to improve TLS since not only the first instruction is used in a trace to fetch a set of instructions from the trace cache. M. Postiff et al. allude to an improvement in the trace selection, also referred to as an indexing mechanism, used by Rotenberg et al., “A more sophisticated indexing mechanism that can access some internal blocks of traces could improve correct-path hit rate” [PTM99].

3.3.10 TSB and Trace Prediction

The next trace predictor is presented in a following section in this chapter; however we will now take the opportunity to highlight how it supports trace search back trace selection. The next trace predictor is used to predict a trace and TSB is used to find an instruction in that trace.

A new trace is constructed from recently graduated instructions. When the new trace index is added to the next trace predictor, recently fetched instruction history and trace history are used to add the new trace index to the predictor. The next trace predictor is speculatively updated based on the results of trace selection when the trace is predicted and passed to the decode latch. It is also updated when branch outcomes or indirect jump targets are mispredicted.

TSB works well with the next trace predictor design because a trace is a set of instructions and it is the atom of prediction for the next trace predictor. TSB needs a trace to search. Once provided a trace, it can select which instruction to begin passing to the

decode latch. If TSB hits then the predictor is updated with the correct prediction thereby training it to provide the correct trace. If it misses then the trace was clearly not valid for the required fetch instruction and the predictor is updated with the incorrect prediction.

3.3.11 Duplication

Duplication is a measure of the amount of wasted storage in the trace cache. An instruction in the trace cache is said to be duplicate if it has the same PC as another in the trace cache and, in the case of a conditional branches or indirect jumps, the same target. We consider conditional branches and indirect jumps with the same PC but different targets to be unique because their execution results in different fetch paths.

Duplicate basic blocks in the trace cache, resulting in wasted cache storage space, are a well known problem. Postiff et al. relate duplication to trace cache efficiency, "Duplication is a measure of how efficiently the 'un-fragmented' storage in the trace cache is used" [PTM99]. Vandierendonck et al. summarize duplication as a type of redundancy, "The first type of redundancy detects multiple copies of instructions and is called duplication" [VRBV02]. We have based our definition on their work.

The equation below defines duplication, where *t* is the total number of instructions present in the trace cache and *u* is the number of unique instructions in the trace cache:

$$d(t, u) = \frac{t - u}{t}$$

An improvement in learning time comes along with the reduction in duplication simply because it takes less time to sequentially insert three new traces into the trace cache than it does to insert nine traces. Thereby requiring less time between when the

first instructions from a new loop are fetched during ILS and when the full loop can be fetched during TLS. In other words, the fact that three traces can fully represent the example loop when using TSB allows ILS to transition to TLS and begin fetching from the trace cache sooner than without TSB.

3.3.12 Fragmentation

Fragmentation is another measure of the amount of wasted storage in the trace cache. Postiff et al. relate trace cache fragmentation to storage utilization, "Fragmentation is a measure of storage utilization which describes the portion of the trace cache that is unused because of traces that are shorter than 16 instructions" [PTM99].

Fragmentation quantifies wasted storage by describing the portion of the trace cache that is unused because of traces that are shorter than their maximum length. The equation given below defines fragmentation, where *i* is the total number of instructions in the trace cache, *m* is the maximum number of instructions in a trace, *w* is the total number of trace cache lines and *u* is the number of unused trace cache lines:

$$f(i, m, w, u) = \frac{i}{m(w-u)}$$

We include the following CTI instructions in traces: conditional/unconditional branches, direct/indirect jumps, calls and returns. This is done to allow the trace fetch stream to run continuously and to minimize trace cache fragmentation.

Our inclusion of most CTI instructions, including indirect jumps, is in contrast to prior trace cache research that was concerned with implementation complications caused by including the basic blocks following CTI's that have indirect targets. The main concern was that doing so would complicate trace ID creation. Rotenberg et al. explain,

"instructions with indirect targets are not embedded to allow traces to be uniquely identified by their starting address and the outcomes of any conditional branches" [RBS96]. The termination strategy employed is to include the CTI with an indirect target in the trace as the final instruction and then stop filling the trace at that point. In the worst case, a trace may hold only a single indirect CTI.

Terminating a trace before the maximum length is reached so as not to include a CTI target basic block is described Rotenberg et al. in the following quote, "the trace cache does not store returns, indirect jumps, or traps at all; the line buffer aborts a fill when it detects any of these instructions" [RBS96]. Also by Jacobson et al., "Any control instruction that has an indirect target can not be embedded in a trace and must be at the end of the trace. This means that some traces will be shorter than their maximum length." [JRS97]. Similarly this is described by Rotenberg et al. in their work that includes a next trace predictor, "Trace selection is constrained only by the maximum trace length of 16 instructions and indirect branches (returns and jump/call indirects) terminate traces" [RBS99].

The inclusion of most CTI target basic blocks in a trace leverages TSB to reduce duplication. The improvement comes from the fact that the target basic block can be fetched even if the CTI that precedes it in the trace is not the same as that specifying the target block in the current fetch instruction stream. TSB can simply select the starting instruction of the target basic block from within the trace and fetch from that point to the end of the trace.

A five bit field storing the outcomes of the first five conditional branches in a trace is stored in each NTP entry and with each TC line. The outcomes of the predicted trace are compared with the actual trace to select a trace. Thus, a trace is selected when

the outcomes and the fetch PC match. During TSB, the outcomes for conditional branches preceding the matching fetch PC in a trace are masked during comparison so as to only compare the outcomes of the conditional branches that will be fetched from the selected trace.

3.3.13 Next Trace Predictor

The *next trace predictor*, we will use the acronym *NTP*, predicts at the rate of one trace per cycle and is used to select a trace from the trace cache to fetch. As the fetch unit attempts to fetch instructions from the instruction cache during ILS, the NTP is simultaneously referenced for the next trace prediction. If the NTP hits and the predicted trace is found in the trace cache then the TC enters TLS.

Our NTP design supports features that are derived from the work of Jacobson et al. [JRS97]. The derivative features are a hybrid NTP, reduced cost predictor design, the intermediate shift register and history backup windows.

The NTP uses a saturating counter in the primary and secondary tables that is based on the design from Jacobson et al. [JRS97]. The primary table incorporates a two bit counter and the secondary uses a four bit counter.

The saturating counter of an entry in the next trace predictor is decremented when the processor detects a branch direction or indirect jump target misprediction. For this reason, each branch and indirect jump issued during trace level sequencing is tagged with the NTP index of the entry and the table used to make the trace prediction. Additionally, the NTP is decremented if it selects a trace from the trace cache and a matching instruction is not found during TSB. The trace predictor is incremented when all of the instructions fetched from a trace in the trace cache are committed.

The NTP's primary table uses a trace cache index based index to predict the next trace. The trace indices that have been predicted previously are shifted into a history register. A hashing scheme is used to create the next trace predictor index that is then used to lookup the next trace prediction in the primary table. The primary prediction table's saturating counter is different than that which is used in a branch predictor in that it is not used to predict direction. It is used to determine the effectiveness of a NTP entry and in the case that it is zero the NTP entry is not used. This prediction scheme is based on the path-based next trace predictor of Jacobson et al. [JRS97].

The primary table's history register is shifted when the NTP primary table is incremented. It is backed up before it is shifted and the backups are stored until all of the trace's instructions are either committed or squashed. The history register is restored to an earlier state in the cases that a NTP primary table entry is decremented. This technique allows the history register to always hold the complete and correct TLS program path after misprediction recovery.

The NTP was chosen because it uses trace history to predict the next trace and has a higher accuracy than MBP based TC designs. The NTP is a cleaner design than that first proposed by Jacobson et al. [JRS96]. The design described in [JRS96] uses a multiple branch predictor to predict a set of branch directions and then compares these to the directions of branches contained in a trace during trace selection. Therefore, it explicitly predicts the direction of branches in a trace. This is in contrast to a next trace predictor based TC fetch unit that implicitly predicts the direction of branches in a trace. Additionally, it is shown that the NTP has higher prediction accuracy than the original TC design [JRS97].

A benefit of our TSB trace selection design is that the PC of the first trace instruction to be fetched from a trace does not need to be stored for a later comparison. This allows us to use a cost-reduced NTP implementation where the NTP does not need to store the full trace ID.

A hybrid predictor configuration is used in our TC design. Our secondary predictor design is based on that proposed by Jacobson et al. [JRS97], however the secondary table does not use the last used trace index to index the table as Jacobson et al. have done, "The secondary predictor requires a shorter learning time and suffers less aliasing pressure." [JRS97].

We have replaced the Jacobson et al. index generation algorithm with one that allows the secondary predictor to be aware of the current instruction fetch path thereby reducing the amount of time spent in ILS. The secondary table's index represents a path history of recently fetched instructions. Our secondary index is generated by XORing the PCs of the most recently fetched *n* instructions, where *n* is equal to the length of a trace. This imbues locality to the secondary table index that helps to improve TLS performance by keeping the next trace predicted by the secondary table relevant to the fetched instruction stream.

Our algorithm permits the next trace predicted by the secondary predictor to be correlated to the fetched instructions from both ILS and TLS. This improves on prior work that uses the last fetch trace ID or index to select a secondary table entry thereby giving the secondary table visibility to only TLS fetched instructions.

In our design, the secondary predictor is used as a technique to help the NTP predict a trace that will allow the fetch unit to transition from ILS to TLS. Once in TLS, the primary table will become more useful since it tracks trace history. Without the

secondary table, trace history is not as helpful in producing traces that are relevant to the instructions being fetched by ILS since it is only cognizant of trace history.

A cost-reduced predictor is used in our TC design. It stores the hashed trace cache index used to lookup a trace. Our cost-reduced design is based on that proposed by Jacobson et al. [JRS97]. In their design, a trace ID read from the NTP table must be hashed to index the trace cache and they use this hashing function to create a cost-reduced predictor. They move the trace ID hashing function to the input side of the NTP to hash the trace ID before it is stored in the table such that the history register stores the trace cache index instead of the trace ID. This cost-reduced NTP design significantly reduces the size of the NTP tables since the trace ID is larger than 30 bits and a typical trace cache index is around ten bits. We have implemented a similar technique and use the trace cache indices as elements in the primary table's shift register.

CHAPTER 4. SIZE AND LATENCY

This chapter discusses specific size and latency considerations for the major data structures used in our fetch unit comparison. The specific data structures presented in this chapter are in the "standard" configuration and any alternate configurations that are used during our analysis will be presented with augmented size and/or latency information as appropriate. If we do not mention a specialized configuration of these data structures then the reader may assume the size and latency calculations presented in this chapter apply to the discussion at hand.

4.1 Latency and Timing of Pipelines

The simulated processor pipelines are typical for a dynamically scheduled RISC processor. Our MBP simulations assume six stages between ID and EX and the TC simulations vary this from five down to three. The pipelines are as follows:

IF | CA AR | ID P1 P2 P3 | P4 P5 P6 EX WB | C

IF = Instruction Fetch (fetched instructions are predicted)

CA = Instruction Cache Lookup

AR = Arrange arriving instructions (No Miss)

ID = Decode starts, reorder buffer slot allocated

P1 to P3 = Decode, rename, and etc.

P4 to P6 = Dispatch stages

EX = Execution

WB = Write Back

C = Commit

The vertical bars in the pipelines diagram given above indicate pipeline transitions that are performed via instruction queues. The pipeline stage before the bar inputs instructions into the queue and the pipeline stage following the bar removes instructions from that queue.

The MBP will need to realign the instructions fetched from multiple instruction cache lines into the decode latch. The TC does not require this step because the trace cache holds instructions in their runtime order. This is the reason we explore a variety of stages between ID to EX for the TC.

4.2 Branch Address Cache

Before discussing the BAC we will define some terminology that has not yet been defined. The BAC stores a *node tree* of basic blocks which is defined as a set of basic blocks that has a tree-like structure. A *node* in the node tree is a basic block and the edges connecting nodes in the tree are defined by the basic block's CTI targets. The node tree has a *root node* that is the first basic block in a tree of basic blocks. The node tree contains all the basic blocks reachable from the root node to a certain distance. A BAC *entry* is composed of the node tree of basic blocks.

The BAC requires a single cycle to access an entry's node tree. It is designed to hash a branch PC of a root node and then use the embedded tag to determine a hit. In the case of a hit the entry is used, otherwise it is ignored.

The *BAC width* is defined as the log base two of the number of entries in the BAC. Since the BAC must store all possible paths from a PC to a certain order it is rather large and contains many entries. The size of the BAC is 2^w entries, where w is the width.

A BAC entry contains a tag field, a one bit valid entry field, and $44(2^d - 1)$ bits used to describe the node tree, where *d* is the order of the BAC [K02, YMP93]. The tag field is equal to the 30 effective PC bits of the root node's CTI subtracted from the BAC width. A BAC entry's node tree is described by $2^d - 1$ block tree node fields that are each 44 bits wide. Each block tree node field is composed of the following sub-fields:

- Effective PC bits of the CTI at the end of the basic block (30 bits)
- Type of the CTI at the end of the basic block (4 bits)
- Number of instructions in the basic block (10 bits)

The number of bits *S* in a BAC of order *d* and a width *w* is given by:

$S(d, w) = 2^w(30 - w + 1 + 44(2^d - 1))$, where *d* and *w* are positive integers

Table 1. BAC Size

d	w	S(d,w)
1	5	2,240
2	5	5,056
3	5	10,688
4	5	21,952
1	6	4,416
2	6	10,048
3	6	21,312
4	6	43,840
1	8	17,152
2	8	39,680
3	8	84,736
4	8	174,848
1	10	66,560
2	10	156,672
3	10	336,896
4	10	697,344
1	12	258,048
2	12	618,496
3	12	1,339,392
4	12	2,781,184
1	14	999,424
2	14	2,441,216
3	14	5,324,800
4	14	11,091,968
1	16	3,866,624
2	16	9,633,792

(table continued)

3	16	21,168,128
4	16	44,236,800
1	18	14,942,208
2	18	38,010,880
3	18	84,148,224
4	18	176,422,912

4.3 YAGS Multiple Branch Predictor

The YAGS multiple branch predictor contains the PBHT, SHTs and control logic. The selection of a YAGS predictor is a good choice for the MBP because it is a global scheme with a good chip area to performance ratio [EM98]. A *global scheme* uses a global history register that records the direction of prior branches to predict the direction of the current branch [M93].

The chosen design is able to make multiple branch predictions in a single cycle and has a number of prediction ports equal to the order of the BAC. As described in the Multiple Branch Predictor section of Chapter 3, it is derived from the work of Eden et al. [EM98] but the design has been extended to predict multiple branch directions in a single cycle.

The PBHT of the YAGS predictor is a straight forward bimodal design and each entry contains a two bit saturating counter used for direction prediction. The direction predicted is used to select one of two tables – the SHTs. The two SHTs are of equal size and each entry contains a ten bit partial tag and a two bit saturating counter.

The number of bits *S* in a YAGS predictor with a PBHT of width *w* and SHT widths *s* is given by:

$S(w, s) = 2^{w}(2)+ 2^{s}(24)$, where *w* and *s* are positive integers

Table 2. YAGS Size

w	s	S(w,s)
2	1	64
4	2	128
6	3	320
8	4	896
10	5	2,816
12	6	9,728
14	7	35,840
16	8	137,216
18	9	536,576

4.4 Trace Cache

The trace cache is designed to lookup a trace in a single cycle. Each trace cache line is composed of a trace record that contains the bits of each instruction contained in the trace and is as long as the trace length. In our design there are 32 instruction bits and we will use *l* as the trace length.

In our design, the length of a trace is constrained to the issue width of the processor. This is done because the maximum number of instructions the fetch unit can issue to decode is the issue width and therefore bounds the peak fetch rate.

The outcomes of the first five conditional branches embedded in a trace are used for trace selection. These five outcome bits are stored in every trace cache entry.

The number of bits *S* of a trace cache with a trace length *l* and a width of *w* is given by:

$S(l, w) = 2^{w}(32*l + 5)$, where *l* and *w* are positive integers

Table 3. Trace Cache Size

l	w	S(l,w)
8	2	1,044
12	2	1,556
16	2	2,068
8	4	4,176
12	4	6,224
16	4	8,272

(table continued)

8	6	16,704
12	6	24,896
16	6	33,088
8	8	66,816
12	8	99,584
16	8	132,352
8	10	267,264
12	10	398,336
16	10	529,408
8	12	1,069,056
12	12	1,593,344
16	12	2,117,632
8	14	4,276,224
12	14	6,373,376
16	14	8,470,528
8	16	17,104,896
12	16	25,493,504
16	16	33,882,112
8	18	68,419,584
12	18	101,974,016
16	18	135,528,448

4.5 Next Trace Predictor

The Next Trace Predictor is designed to provide a single trace prediction each cycle. The NTP primary table and secondary table entries contain an index to an entry in the trace cache. The primary table has a two bit saturating counter and the secondary table has a four bit counter used for prediction.

The outcomes of the first five branches embedded in the trace that is referenced by an entry in the NTP are used for trace selection. Each entry in the primary and secondary table has a five bit field to hold these outcomes.

The number of bits *S* in a NTP with a primary table width *s*, trace cache index width *w* and secondary table width *y* is given by:

$S(s, w, y) = 2^s(2 + w + 5) + 2^y(4 + w + 5)$, where *w*, *s* and *y* are positive integers.

Table 4. Next Trace Predictor Size

s	w	y	S(s,w,y)
2	8	2	128
2	12	2	160
2	16	2	192
4	8	3	376
4	12	3	472
4	16	3	568
6	8	5	1,504
6	12	5	1,888
6	16	5	2,272
8	8	6	4,928
8	12	6	6,208
8	16	6	7,488
10	8	7	17,536
10	12	7	22,144
10	16	7	26,752
12	8	9	70,144
12	12	9	88,576
12	16	9	107,008
14	8	10	263,168
14	12	10	332,800
14	16	10	402,432
16	8	11	1,017,856
16	12	11	1,288,192
16	16	11	1,558,528
18	8	13	4,071,424
18	12	13	5,152,768
18	16	13	6,234,112

4.6 TC Single Branch Predictor

The trace cache single branch predictor is a single cycle implementation. It creates an index by masking the effective bits of the to-be-predicted branch PC to the width of the BHT. The BHT's saturating counter is two bits and is used for prediction.

The number of bits *S* in a BHT of width *m* is given by:

$S(m) = 2^m (2)$, where *m* is the width of the BHT

Table 5. Trace Cache Branch Predictor Size

m	S(m)
2	8
4	32
6	128
8	512

(table continued)

12	8,192
14	32,768
16	131,072
18	524,288

4.7 Target Buffer

The target buffer is a direct mapped cache that holds CTI target PC addresses for lookup using the masked effective bits of the CTI PC. Only the 30 effective bits of the target PC must be stored.

The target buffer uses a tag to verify that the predicted target belongs to the CTI used for lookup. The tag is equal to the CTI PC's 30 effective bits minus the width of the index since comparing these would be redundant.

The number of bits *S* in a target buffer of width *n* is given by:

$S(n) = 2^n (30 – n + 30)$, where *n* is the width of the target buffer

Table 6. Trace Cache Target Buffer Size

n	**S(n)**
2	232
4	896
6	3,456
8	13,312
12	196,608
14	753,664
16	2,883,584
18	11,010,048

4.8 MBP Multi-Ported L1 Instruction Cache

The multi-ported instruction cache of the MBP is able to fetch more than one line per port per cycle and has a number of ports equal to the order of the BAC. We will use the acronym *IC* for the instruction cache.

The size and internal structure of the multi-ported cache is the same as the trace cache's single ported IC in our analysis. Please refer to the next section describing that cache for an analysis of the IC's size.

4.9 TC Singly-Ported L1 Instruction Cache

The TC's L1 instruction cache is singly ported. It is four way set associative with a LRU replacement policy.

The number of bits *S* of an IC with a line length *i* and an IC width *b* is given by*:*

$S(i, b) = 2^{b} (1 + 30 - b + 8i)$, where *i* and *b* are positive integers.

Table 7. Instruction Cache Size

i	B	S(i,b)
64	2	2,164
64	4	8,624
64	6	34,368
64	8	136,960
64	10	545,792
64	12	2,174,976
64	14	8,667,136
64	16	34,537,472
64	18	137,625,600

CHAPTER 5. IMPLEMENTATION

This chapter explains our simulator implementation and discusses the features of our timing based dynamically scheduled superscalar processor simulation outside of the fetch unit design.

5.1 RSIML

RSIML is derived from RSIM, which was developed at Rice University and at some point was moved to the University of Illinois for continued development and maintenance. RSIM simulates shared-memory multiprocessors built from processors that aggressively exploit instruction-level parallelism. RSIM is execution-driven and models state-of-the-art ILP processors, an aggressive memory system, and a multiprocessor coherence protocol and interconnect, including contention at all resources. It is a dynamically scheduled processor simulation [UI04].

The RSIM processor simulation features:

Multiple instruction issue

Out-of-order scheduling

Register renaming

Dynamic branch prediction

Non-blocking loads and stores

Speculative load execution before prior stores are disambiguated

Optimized memory consistency implementations

The memory simulation features:

Two-level cache hierarchy

Multiported and pipelined cache, pipelined L2 cache

Multiple outstanding cache requests

Memory interleaving

Software-controlled non-binding prefetching

RSIM simulates a processor using a sub-set of the SPARC V8 instruction set architecture [WG94]. RSIM benchmark programs are compiled for Solaris and for the work done here Solaris 2.7/2.8.

RSIML is derived from the RSIM distribution. The derivation began in the late 1990's in the Electrical and Computer Engineering department at Louisiana State University. RSIM is an open source distribution that supported many of the state of the art ILP exploitations required to simulate at a timing level multiple branch prediction architectures. Modifications are made to the simulator to simulate multiple branch prediction, move the simulator from a SPARC based Solaris host to an X86 based LINUX host and to improve performance reporting [K02].

We have further modified RSIML to simulate a trace cache fetch unit architecture and added additional reporting as necessary to gather statistics on the trace cache.

5.2 Supermike and Data Processing

Supermike is Beowulf class supercomputer with 4 head nodes and 512 compute nodes capable of over 2 Teraflops. Each compute node has dual Intel P4 Xeon processors with a clock speed of over 3 gigahertz and 2 gigabytes of memory. Each compute node has a 10/100 based Ethernet and a PCI based Myrinet card. The Myrinet network comprises the data plane and allows multi-gigabit transfers between each compute node.

The file storage system has 800GB of IBM ESS Model 800 San Storage via Fiber Channel, 2 terabytes of Raid 5 Storage and 80 gigabytes of IDE storage per compute node.

Supermike is managed by LSU's Center for Computation and Technology and funded through Louisiana legislative appropriations.

We leverage Supermike to allow a deep and broad study of the simulation parameter space. The simulations are run in parallel, one for each processor. Statistical data is collected by each simulation and stored in the file system for post analysis. A suite of Perl scripts is used to interpret a parameter file and issue the parallel simulations as a batch job to the cluster through the PBS service.

CHAPTER 6. SIMULATION PARAMETERS

This chapter discusses the simulated processor's parameters and benchmarks used to produce our results. The analysis of the effect of the major structures in the MBP and TC fetch unit simulations were performed by holding all parameters constant at the highest value listed in the parameter tables and then varying one parameter across the listed range.

6.1 Benchmarks

The simulated executables are built from the SPEC CPU2000 suite and use reduced input sizes to reduce simulation time. We simulate mostly integer programs since they have a smaller basic and non-sequential block size than floating point simulations and therefore should more readily benefit from the fetch unit architectures under study. We do simulate a floating point program to see the effect of the fetch unit architectures under study.

Our simulations are limited to committing at most fifteen million instructions to reduce simulation times and allow a broader study of the parameter space. All of the executable programs simulated were compiled with the optimizations provided in the SPEC CPU2000 suite and targeted to an UltraSPARC II processor.

The following integer benchmarks are used for our comparison:

1. perl – An interpreter for the Perl language
2. TeX – Document formatter/typesetter
3. gcc (cc1) - GNU C compiler
4. gzip – A Lempel-Ziv coding (LZ77) file compressor
5. bzip2 - A block-sorting file compressor

The following floating point simulation is used for our comparison:

1. swim - Shallow Water Modeling

6.2 MBP and TC Parameters

Please refer to Appendix A where we define terminology and acronyms used in the tables presented in this section. The plotted results of all simulation runs are presented in Appendix B.

Table 8 details the multiple branch predictor parameters that are unique.

Table 8. Unique Multiple Branch Predictor Parameters

Parameter	Size
ID to EX Stages	6
YAGS Primary Branch History Table Width	5,6,8,10,12,14,16,18
BAC Order	1,2,3,4
BAC Width	2,4,6,8,10,12,14,16,18

The table given below details the trace cache parameters that are unique.

Table 9. Varied Trace Cache Parameters

Parameter	Size
Branch History Table Width	2,4,6,8,10,12,14,16,18
NTP Width (Primary Table)	2,4,6,8,10,12,14,16,18
NTP Width (Sec. Table)	2,3,5,6,7,9,10,11,13
Trace Cache Width	2,4,6,8,10,12,14,16,18
ID to EX Pipeline Depth (Stages)	3,4,5

Table 10 details the important parameters common to both the multiple branch predictor and the trace cache.

Table 10. Common Parameters

Parameter	Size
Decode Width	8,12,16
Instruction Cache Width	2,4,6,8,10,12,14,16,18
L1 Cache Latency	1
L1 Line Length (bytes)	64
L1 and L2 Associativity	4
L2 Width	12
L2 Cache Latency	10
Memory Latency	100
Memory Interleaving	16
Reorder Buffer Depth	256 for decode widths 8 & 12; 512 for decode width 16
Number of FPUs	= Decode Width / 2
Number of ALUs	= Decode Width

6.3 Equal Cost Parameters

We first ran the MBP and TC parameters variations to determine the values that give a reasonable cost versus performance trade-off. Then we picked the parameters we think perform the best for the cost for the MBP and TC. Finally, we adjusted the parameters until we produced a close match in implementation size for the sum of the various data structures within each fetch unit design.

We use the formulae presented in the SIZE AND LATENCY chapter to compute the implementation sizes.

Table 11 lists the parameters used for our study of equal cost implementations. If a parameter is not listed then it is assumed to be the same as what is presented in the common MBP and TC parameters table.

The TC implementation size is 3.3% smaller than the MBP implementation under the parameters we have chosen.

Table 11. Table of Equal Cost Parameters and Implementation Sizes

Parameter Name	TC	Size in Bits	MBP	Size in Bits
YAGS Primary Branch History Table Width	N/A	N/A	14	35,840
BAC Order	N/A	N/A	3	N/A
BAC Width	N/A	N/A	13	2,670,592
Trace Cache Branch History Table Width	14	32,768	N/A	N/A
NTP Table Width (Primary / Secondary)	10/7	26,752	N/A	N/A
Trace Cache Target Buffer	13	385,024	N/A	N/A
Trace Cache Width	10	529,408	N/A	N/A
ID to EX Pipeline Depth (Stages)	4	N/A	6	N/A
Issue Width	16	N/A	16	N/A
Instruction Cache Width	12	2,174,976	10	545,792
Total Implementation Size		**3,148,928**		**3,252,224**

A decode width of 16 is chosen for the equal cost comparisons since prior work focuses on both the MBP and TC simulations with 16-way superscalar processors and we want our results to be comparable. Additionally, the effects of the fetch unit designs on performance are more pronounced.

A BAC order of three is chosen for the MBP due to the fact that it is the best overall value for BAC order based on our results. The L1 IC width of ten for the MBP is chosen because the IPC performance does not significantly improve above this value for the benchmarks studied. A Y-MBP width of fourteen and BAC width of thirteen are picked because they are the best trade-off for size and performance.

An ID to EX stages of four is picked for the TC because it is two stages shorter than the MBP simulation. Two stages shorter pipeline latency is reasonable due to the realignment logic required by the MBP.

The TC branch history table width of fourteen is chosen because when the width is increased above this amount the IPC performance is only slightly affected. The target buffer width of thirteen is picked because it provides a decent performance for a reasonable amount of storage. A NTP width of ten/seven and a TC width of ten are

picked since larger values do not significantly improve IPC performance and to allow for a larger L1 IC size. The TC has a larger L1 instruction cache width than the MBP. A L1 instruction cache width of twelve is chosen due to the fact that the TC's IPC performance improvement due to instruction cache width increases at a fast rate until that point.

CHAPTER 7. ANALYSIS AND COMPARISON

This chapter discusses the results of our simulated multiple branch predictor and trace cache fetch unit designs. We first provide a comparison of the effect of common parameters between the two fetch unit designs, and then analyze the effect of parameters that are specific to each design. Finally, we present an analysis of MBP and TC simulations with an equal implementation cost.

The following chapter draws generalized conclusions on the detailed analysis of the results presented in this chapter. Please refer to the next chapter for conclusions based on the simulated benchmark results.

7.1 Comparison: Effect of Issue Width

The *issue width* is the maximum number of instructions that can be issued in a single cycle. Changing the issue width of the processor effects the MBP and TC differently across the benchmarks but the difference is not dramatic. Please refer to figures 9 through 12 for MBP issue width and figures 25 through 27 for TC issue width.

The table below shows the standard deviation of IPC as a result of varying issue width across the benchmarks under the MBP and TC.

Table 12. Issue Width IPC Standard Deviation

TeX	gcc	swim	perl	gzip	bzip2	Mean Std. Dev.	
0.058	0.043	0.096	0.089	0.175	0.131	0.099	**MBP Standard Deviation**
0.119	0.092	0.082	0.056	0.155	0.042	0.091	**TC Standard Deviation**

The table given below shows the mean percentage of IPC performance increase for the simulated benchmarks as the issue width is increased.

Table 13. Issue Width IPC Mean Percentage Increase

TeX	Gcc	swim	perl	gzip	bzip2	Issue Width Difference	Fetch Type
0.015%	0.018%	0.004%	0.024%	0.022%	0.035%	Δ(8,12)	MBP
0.002%	0.006%	0.033%	0.008%	0.062%	0.016%	Δ(12,16)	MBP
0.043%	0.026%	0.017%	0.056%	0.058%	0.066%	Δ(8,12)	TC
0.067%	0.012%	0.024%	0.008%	0.066%	0.041%	Δ(12,16)	TC

From table 12, we can see that the mean standard deviation is slightly higher for the MBP than the TC. However, the two are very close in the overall amount of change that occurs as the issue width is increased. Table 13 shows that the overall mean percentage increase in IPC performance as issue width is increased is higher overall for the TC than for the MBP.

The MBP TeX benchmark slightly decreases (<1%) the IPC performance for the MBP orders of one with issue widths of eight to twelve, order two with issue widths twelve to sixteen and three with issue widths twelve to sixteen. For all other benchmarks under the MBP and TC simulations the increase in issue width increases the IPC performance.

The changes in MBP order, as the number of BAC nodes is held constant, only slightly modify the overall effect of increases in issue width for the MBP benchmarks. Similarly, the changes in ID to EX stages only slightly modify the overall effect of increases in issue width for the TC benchmarks.

7.2 Comparison: Effect of L1 Instruction Cache Width

The *L1 instruction cache width* is the log base 2 of the number of lines in the L1 instruction cache. Increasing the size of the L1 instruction cache has a more dramatic effect on IPC performance for the TC than for the MBP. After the cache size is greater than 1024 entries, the MBP IPC improvement is small. However, the TC continues to see

significant improvement in IPC until a size of 4096 entries. Please refer to figures 13 through 16 for MBP data and figures 28 through 30 for TC data.

The table below shows the standard deviation of IPC as a result of varying instruction cache sizes across the benchmarks under the MBP and TC.

Table 14. L1 Instruction Cache IPC Standard Deviation

TeX	gcc	swim	perl	gzip	bzip2	Mean Std. Dev.	
0.191	0.110	0.026	0.244	0.159	0.164	0.149	**MBP Standard Deviation**
0.787	0.714	1.474	0.842	0.784	0.735	0.889	**TC Standard Deviation**

From the table above it is clear that varying the size of the L1 instruction cache has a more pronounced effect on the TC IPC performance than it does on the MBP.

The result graphs for the TC simulations show that the gzip benchmark IPC slightly decreases as the L1 IC width increases from the mid-range to the larger values. All of the other benchmarks for the MBP and the TC tend to increase as the width is increased.

The MBP order does affect the rate of IPC performance change for the smaller L1 IC widths. The MBP order four data is markedly different than the others for the smaller cache widths.

7.3 MBP Analysis: Effect of Branch Address Cache Width

The *BAC width* is the log base two of the number of entries in the branch address cache. Varying the BAC width has a unique effect on IPC performance on the benchmarks. In general, an increase in BAC width improves the IPC performance. Please refer to figures 21 through 24 for BAC width data.

As one would expect, smaller BAC sizes cause the BAC to not have enough entries to support the control flow of the fetch instruction stream and this causes a

significant slow down. In general, when the size of the BAC is increased to 64K entries the IPC performance tends to level off for the benchmarks studied.

The table below shows the standard deviation of IPC as a result of varying the BAC width across the benchmarks under the MBP.

Table 15. BAC Width IPC Standard Deviation

TeX	gcc	swim	perl	gzip	bzip2	Mean Std. Dev.	
0.225	0.174	0.077	0.295	0.257	0.226	0.209	**MBP Standard Deviation**

From the table above we can see that swim is affected the least and perl is affected the most by varying the BAC width.

Varying BAC width has an interesting effect on the IPC performance of the perl benchmark because at smaller sizes the rate of increase is moderate, then for the midrange of BAC widths the IPC performance is more strongly changed, finally the IPC performance improvement is slight for the largest BAC widths.

The IPC performance of the benchmarks TeX and bzip2 varies by the same amount and in a similar way based on the result plots and the above standard deviation table.

The gzip benchmark is slightly affected by the two smallest BAC widths and then the IPC quickly improves as the width is increased to ten. After ten the IPC only slightly improves for all BAC orders except four, which takes until a width of twelve to slow its performance improvement.

Increasing BAC width from the smaller to the middle range of values has a moderate affect on the performance of gcc. The performance improvement tapers off for the higher values.

The MBP order has an effect on the IPC results of the benchmarks as the BAC width is increased. The amount of increase is greater for successively larger MBP orders. Additionally, the IPC performance of the benchmarks takes longer to level off for successively larger MBP orders.

7.4 MBP Analysis: Effect of Y-MBP Width

The *Y-MBP width* is the log base two of the number of entries in the YAGS Multiple Branch Predictor. Varying this parameter has a strong effect on the IPC performance of the benchmarks. Please refer to figures 17 through 20 for Y-MBP data.

For this parameter, an interesting effect is that as Y-MBP width is increased from the mid-range through the larger values the IPC performance decreases for some of the benchmarks.

The table below shows the standard deviation of IPC as a result of varying the Y-MBP width across the benchmarks under the MBP.

Table 16. Y-MBP Width IPC Standard Deviation

TeX	gcc	swim	perl	gzip	Bzip2	Mean Std. Dev.	
0.309	0.223	0.202	0.375	0.346	0.250	0.284	**MBP Standard Deviation**

The table above shows that swim was the least affected and perl was the most affected by varying the Y-MBP width. From the standard deviation of the IPC data, it is clear that the benchmarks are more strongly affected by changes in the Y-MBP width than by varying BAC width. In particular, swim is significantly more affected by varying the Y-MBP width than by varying the BAC width.

From the plotted results, we see that the IPC of bzip2, gcc and TeX all peak at the upper middle range of Y-MBP width values and then the IPC decreases toward the high end of values.

Perl has the same “double hump” appearance in its IPC curve as it does for BAC width. The rate of increase in IPC is slight for the smaller Y-MBP widths, increases from the middle widths and finally tapers off for the larger width values.

The MBP order affects the Y-MBP width in a similar way as it does with the BAC width. The amount of increase is greater for successively larger MBP orders. Additionally, the IPC performance of the benchmarks takes longer to level off for successively larger MBP orders.

7.5 MBP Analysis: Effect of Branch Address Cache Order

Varying the BAC order has an effect on IPC performance of the benchmarks but it is not as pronounced as varying the BAC width, Y-MBP width, or L1 instruction cache width.

Figure 3 shows the mean IPC across all varied parameters for the MBP for each benchmark as a function of the BAC order parameter.

The IPC performance of the swim benchmark is only slightly improved by increasing the BAC order. The benchmarks TeX, gcc, swim, and perl all increase monotonically as BAC order increases. The IPC performance of the gzip and bzip2 benchmarks increases through a BAC order of three and then decreases at a BAC order of four.

Table 17 below shows the difference of IPC values across the BAC orders studied.

Table 17. IPC Difference Table for BAC Order

TeX	gcc	swim	perl	gzip	Bzip2	
0.168	0.087	0.026	0.168	0.167	0.165	**Δ(2,1)**
0.032	0.011	0.000	0.032	0.004	0.019	**Δ(3,2)**
0.006	0.007	0.006	0.004	-0.017	-0.003	**Δ(4,3)**

The greatest overall change in IPC performance results from going from BAC order one to two. From a value of two to three there is a less significant change. Finally, from three to four the IPC performance increase is very slight and in fact is a slight decrease from gzip or bzip2.

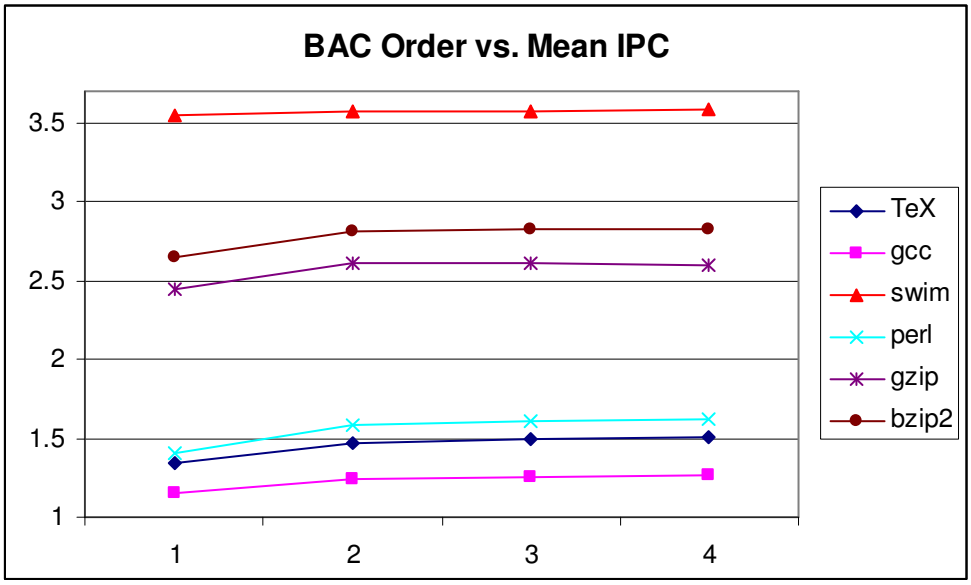

Figure 3. BAC Order vs. Mean IPC

7.6 TC Analysis: Effect of Single Branch Predictor Width

The *single branch predictor width* is the log base two of the number of entries in the predictor table. Varying the SBP width has a very strong effect on the benchmark IPC performance. Please refer to figures 31 to 33 for data on the single branch predictor.

It is of note that there is noise in the smaller values of the table width for the bzip2 and gzip benchmarks. The noise tends to increase as the ID to EX stages decrease from five to three.

The table below shows the standard deviation of IPC as a result of varying the single branch predictor width across the benchmarks under the TC.

Table 18. Single Branch Predictor Width IPC Standard Deviation

TeX	gcc	swim	perl	Gzip	bzip2	Mean Std. Dev.	
0.380	0.260	0.096	0.531	0.199	0.378	0.307	**TC Standard Deviation**

From the table above it is clear that perl is most affected and swim is least affected by increases in the table width.

The benchmarks' IPC tends to increase as the table width is increased. Noise in the larger table values for TeX, swim, gzip and bzip2 cause the IPC performance to decrease at certain points but overall the performance tends to increase.

The ID to EX stages has an effect on the IPC performance as the table width is varied. The effect is an increase in noise for the smaller values of gzip and bzip2 as noted above. The noise increases as the ID to EX stages are decreased from five to four and then the noise decreases as the ID to EX stages further decreases from four to three. Also, the ID to EX stages value of four creates the greatest variation of IPC performance as the single branch predictor width is varied.

7.7 TC Analysis: Effect of Target Buffer Width

The *target buffer width* is the log base two of the number of entries in the target buffer. In general, increasing the buffer width causes the benchmarks' IPC performance to increase. However, the effect is less pronounced than the effect of varying the single

branch predictor table width. Please refer to figures 34 through 36 for data on the target buffer.

The table below shows the standard deviation of IPC as a result of varying the target buffer width across the benchmarks under the TC.

Table 19. Target Buffer Width IPC Standard Deviation

TeX	gcc	swim	perl	Gzip	bzip2	Mean Std. Dev.	
0.114	0.159	0.023	0.156	0.097	0.114	0.110	**TC Standard Deviation**

From the table above we see that gcc was the most affected by increasing the target buffer width and swim was the least affected.

For the larger target buffer widths, the IPC performance of the gzip benchmark tends to moderately decrease (~1%). All other benchmarks' IPC performance tends to increase as the buffer width is increased.

The ID to EX stages has an effect on IPC performance when the target buffer width is varied. The gzip benchmark has noise in the mid-range buffer width values for an ID to EX stages value of four that is not seen for the other ID to EX values. Also, as the ID to EX stages parameter is decreased, the amount of variation in the IPC performance of the benchmarks increases while varying the target buffer width.

7.8 TC Analysis: Effect of Trace Cache Width

The *trace cache width* is the log base two of the number of traces in the trace cache. Increasing the TC width causes an increase in IPC performance for the benchmarks studied. Please refer to figures 40 through 42 for TC width data.

The TC width affects the IPC performance most significantly for the smaller widths. The performance improvement levels out for the mid-range and higher values. It

is interesting to note that a TC width of ten or greater shows negligible improvement of IPC performance for the benchmarks studied.

The table below shows the standard deviation of IPC as a result of varying the trace cache width across the benchmarks.

Table 20. Trace Cache Width IPC Standard Deviation

TeX	gcc	swim	perl	Gzip	bzip2	Mean Std. Dev.	
0.115	0.087	0.013	0.100	0.088	0.115	0.086	**TC Standard Deviation**

From the table above we see that varying the TC width most greatly affects the TeX and bzip2 benchmarks and has the smallest affect on the swim benchmark.

The gzip simulation peaks in IPC performance for the mid-range TC widths and then slightly decreases in performance for the larger values. For all other benchmarks, increasing the TC width increases the IPC performance.

Changing ID to EX stages has a consistent effect on the IPC performance as the TC width is varied.

7.9 TC Analysis: Effect of the Next Trace Predictor Width

The *next trace predictor width* is the log base two of the number of NTP primary table's entries. The secondary table's size is a function of the primary table and is approximately two thirds of the size. Increasing the NTP width causes the IPC performance to increase for the benchmarks studied. Please refer to figures 37 through 39 for NTP width data.

Just as in the case of the TC width, the NTP width affects the IPC performance the most for the smaller values and tends to level off for the mid-range to larger NTP widths.

The table below shows the standard deviation of IPC as a result of varying the next trace predictor width across the benchmarks under the TC.

Table 21. Next Trace Predictor Width IPC Standard Deviation

TeX	gcc	swim	perl	Gzip	bzip2	Mean Std. Dev.	
0.122	0.088	0.012	0.118	0.137	0.122	0.100	**TC Standard Deviation**

From the table above we see that the swim benchmark's IPC performance is least affected by varying the NTP width and TeX and bzip2 are most affected.

All of the benchmarks' IPC performance tends to improve as the NTP width is increased.

The ID to EX stages parameter changes the performance of the benchmarks for the TC width range studied. As the ID to EX stages decrease, there is a decrease in the amount of performance increase due to the NTP width increasing.

7.10 TC Analysis: Effect of the Number of ID to EX Stages

ID to EX stages has an affect on the IPC performance of the benchmarks studied. All of the benchmarks demonstrate an improvement in IPC when the value of the ID to EX stages parameter is reduced.

The graph below shows the mean IPC across all varied parameters for the TC for each benchmark as a function of the ID to EX stages parameter.

From table 22 and figure 4 we see that the increase in ID to EX stages significantly improves the IPC performance of the bzip2, TeX, gzip and perl benchmarks. The IPC performance of the swim and gcc benchmarks is only marginally improved as the ID to EX stages are decreased.

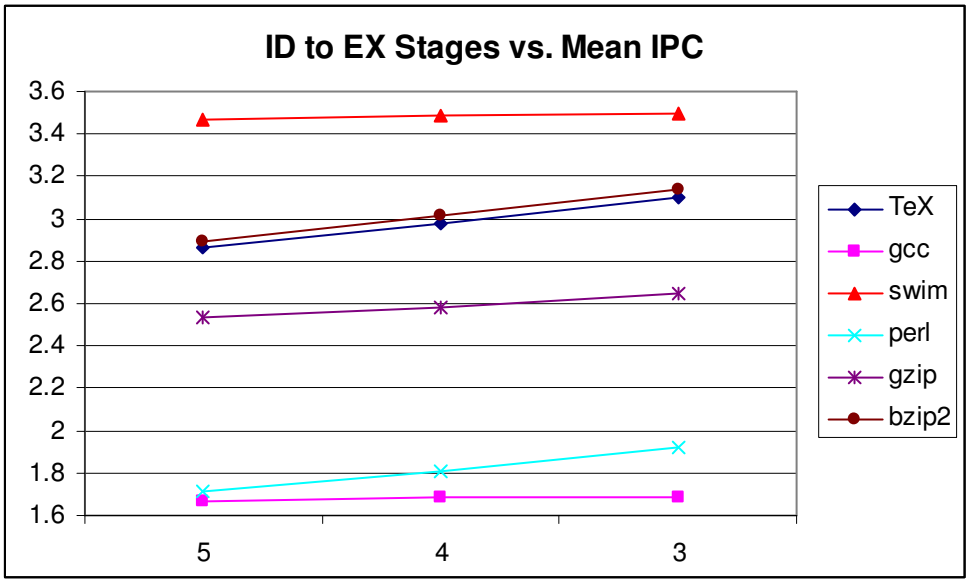

Figure 4. ID to EX Stages vs. Mean IPC

The table below shows the difference of IPC values across the ID to EX stages studied.

Table 22. IPC Difference Table for ID to EX Stages

TeX	gcc	swim	perl	Gzip	bzip2	
0.116	0.014	0.016	0.088	0.046	0.123	**Δ(5,4)**
0.119	0.005	0.016	0.113	0.068	0.122	**Δ(4,3)**

In general, the improvement in IPC for the ID to EX decrease from four to three is more pronounced than for the change from five to four. The gcc benchmark is an exception and shows the opposite trend. The change is approximately the same from the bzip2 benchmark but is slightly greater from five to four than from four to three.

7.11 Equal Cost Comparison Simulations

In this section we discuss the results of the equal cost implementation comparison for the MBP and TC simulations across the benchmarks. The parameters used were chosen and the size of the implementations was computed in the Equal Cost Parameters

section of Chapter 6. In summary, the implementation size of the TC is approximately 3.3% smaller than the MBP and the data structure sizes were chosen to have the greatest IPC performance based on our simulation results.

The following graph plots the IPC performance of the equal cost implementation TC and MBP simulations for the benchmarks under study.

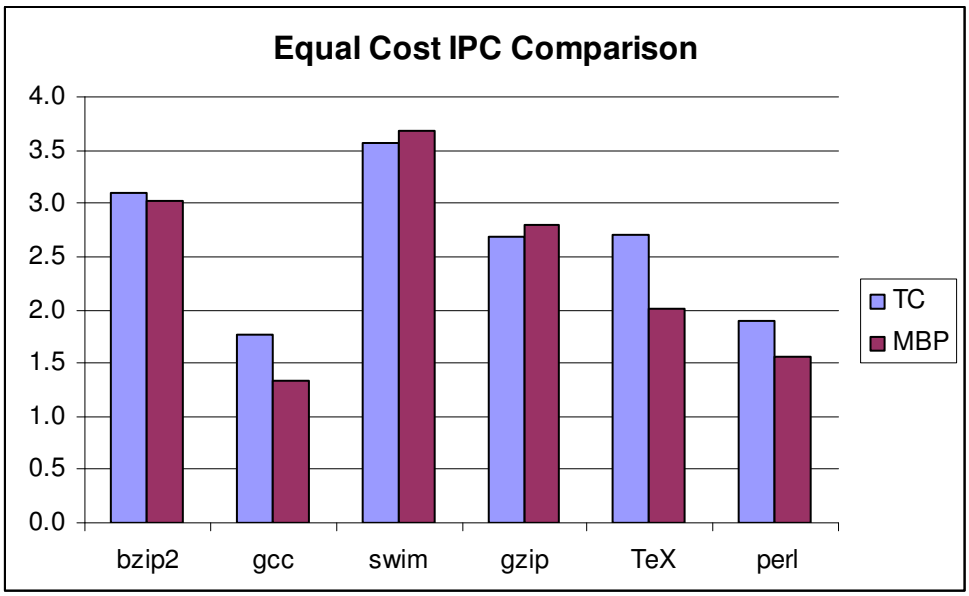

Figure 5. Equal Cost IPC Comparison

From the graph above, we see that the MBP out-performs the TC for the swim and gzip benchmarks. The TC out-performs the MBP for the TeX, gcc, bzip2 and perl benchmarks.

Figures 6 and 7 show the re-order buffer slot utilization for the MBP and TC fetch unit simulations studied across the benchmarks. The 100% stacked column plots compare the relative number of ROB slots for when the ROB was full, instructions were squashed and instructions were committed.

Figure 8 compares the accuracy of instruction fetch speculation between the two fetch unit architectures. The *speculation accuracy* is the ratio of graduated instructions to the number of decoded instructions expressed as a percentage.

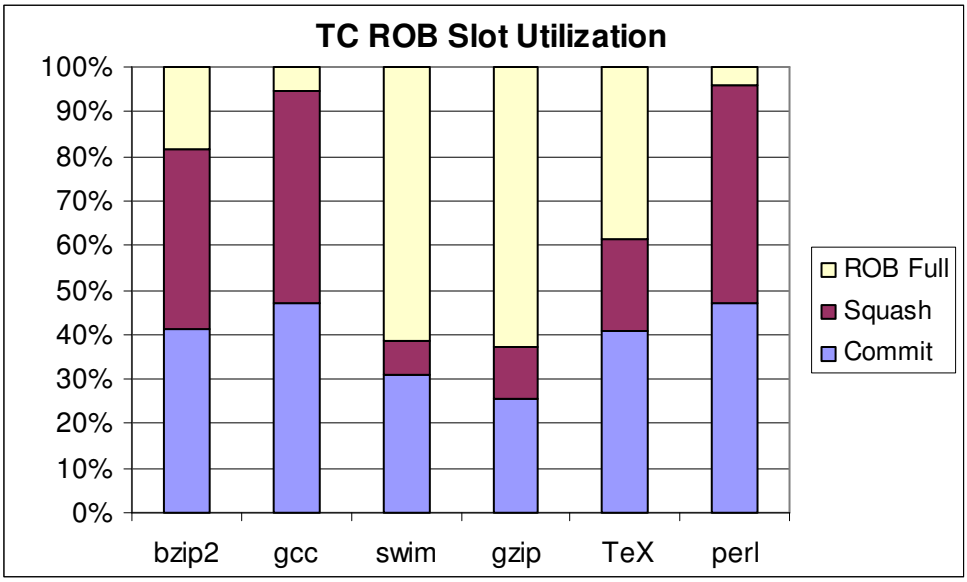

Figure 6. TC ROB Slot Utilization

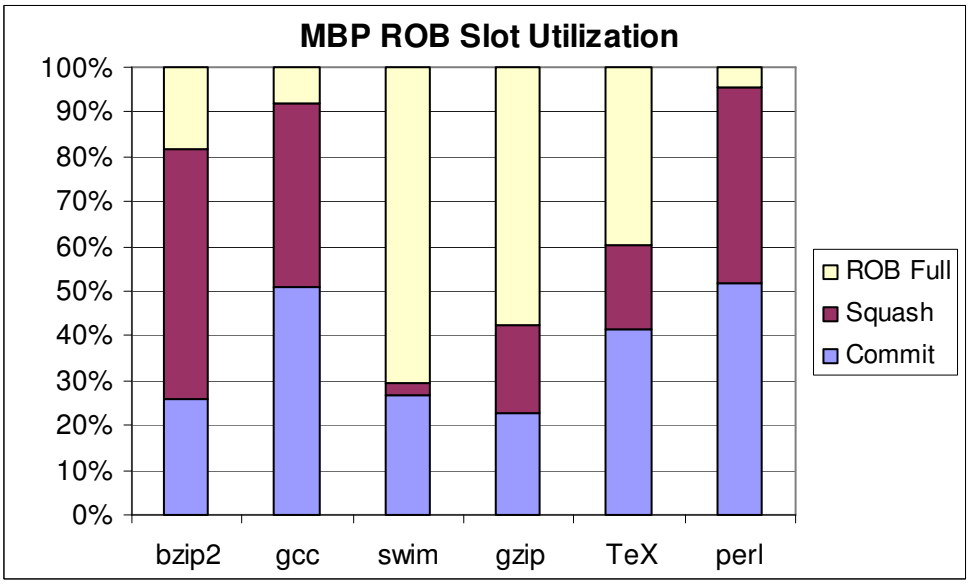

Figure 7. MBP ROB Slot Utilization

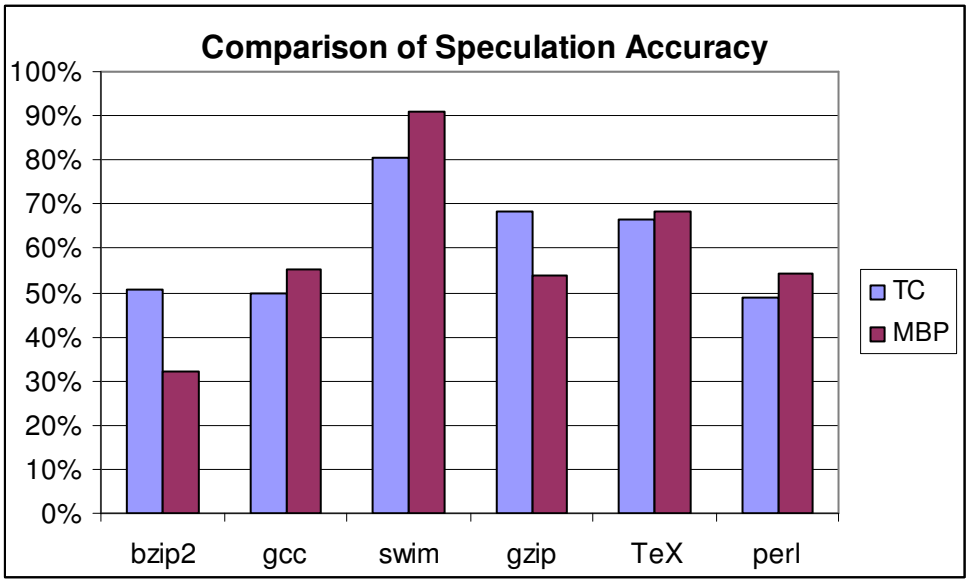

Figure 8. Comparison of Speculation Accuracy

The ROB full slot utilization of bzip2 for the TC is approximately the same for TC and the MBP as seen in figures 6 and 7. However, the number of squashed instructions for the MBP is significantly greater for the MBP than for the TC. The fact that relatively more instructions were committed by the TC contributes to the greater IPC performance of bzip2 for the TC in comparison to the MBP in figure 5. Figure 8 supports our conclusion by showing that the TC is more accurate in speculative fetching of instructions.

Figure 8 shows greater speculation accuracy for the MBP than the TC for the gcc benchmark. Figures 6 and 7 demonstrate a smaller percentage of full ROB slots for the TC than the MBP. Interestingly, figure 5 shows that the TC out-performs the MBP. In this case, the shorter pipeline assists the TC's higher IPC performance due to the fact that it allows a faster misprediction recovery. A larger ROB may help the MBP to improve the IPC performance relative to the TC since the ROB was full more often for the MBP.

Figures 6 and 7 indicate that the MBP has a higher ROB full percentage than the TC for the swim benchmark. Figure 8 shows significantly greater speculation accuracy for the MBP than the TC. These factors contribute to the higher IPC performance of the MBP in figure 5.

The gzip benchmark has a greater IPC performance for the MBP simulation than for the TC as seen in figure 5. However, from figure 8 we see that the TC has significantly greater speculation accuracy than the MBP. Figures 6 and 7 help to explain this inconsistency between IPC performance and speculation accuracy since the TC's ROB full percentage is larger than the MBP's preventing the TC from fetching instructions. A larger size ROB may help to lower the ROB full percentage of the TC and improve its IPC performance relative to the MBP.

For the TeX benchmark, figures 6 and 7 establish that the TC and MBP have approximately the same percentages of ROB slot utilization. Figure 8 demonstrates close speculation accuracy. Even though these results are close the TC greatly out-performs the MBP. This can be explained by the shorter pipeline of the TC allowing faster recovery from CTI misprediction.

Figure 5 demonstrates that the TC has a greater IPC performance than the MBP for the perl benchmark. The TC and MBP have approximately equal percentage of ROB full slots from figures 6 and 7. Figure 8 establishes that the MBP has greater speculation accuracy than the TC. For this benchmark, the shorter pipeline of the TC most likely improves the CTI misprediction recovery time for the TC and results in higher IPC performance.

CHAPTER 8. CONCLUSIONS

The simulations studied in this work have provided insight into the performance characteristics and trade-offs of the multiple branch predictor and trace cache fetch unit architectures. The MBP and TC both demonstrate the ability to take advantage of fetching multiple basic blocks per cycle to improve the IPC performance of the benchmarks studied.

8.1 Multiple Branch Predictor Fetch Unit Design

The branch address cache performs well in our simulations. The IPC performance benefit of the MBP is seen with a BAC order of two. Increasing the BAC width to gain performance is a better use of chip area compared to deepening the BAC order over two. A width of thirteen for the BAC provides good IPC performance for the storage required in its implementation.

The Y-MBP predictor performs well and it has a significant impact on performance. From our results, picking a Y-MBP width of fourteen provides the best IPC performance to chip area trade-off.

Studies of designs that allow shortening of the MBP fetch unit pipeline are warranted. Designs such as the collapsing buffer [CMMP95] help to shorten the realignment logic.

An interesting research MBP fetch unit based system might implement a dual-ported trace cache to hold up to two basic blocks and a multiple branch predictor with an order two branch address cache used to fetch from the trace cache or the dual-ported L1 instruction cache. The proposed design is based on that used by Rotenberg et al. [RBS96], but more than one trace can be fetched per cycle and a trace is permitted to be

combined with a line from the instruction cache. This system may reduce pipeline stages required for realignment while fetching up to four basic blocks per cycle.

8.2 Trace Cache Fetch Unit Design

The trace cache in our simulations is able to take advantage of the TSB selection technique. It also is able to benefit from the trace termination policy. These together allow a moderately sized trace cache to take advantage of the dynamic instruction stream. This is seen in the results where a trace cache size of 1024 entries provides a competitive IPC performance across the benchmarks studied. Using this TC size, it is able to outpace the MBP in four of the six benchmarks studied with an approximately equal implementation cost.

Associativity in the trace cache may result in improved performance. The IPC improvement gained from increases in the trace cache size were modest and this points to room for improvement in the TC design. Prior research by Rotenberg et al. suggests that increasing cache associativity improves IPC performance as the cache size is increased [RBS99]. The inclusion of a tag in the trace cache, as done by Rotenberg et al., would increase its size but may improve IPC performance.

A higher performance single branch predictor and target buffer for the TC fetch is justified by the correlation between their sizes and the IPC performance of the benchmarks. A GSHARE or YAGS branch predictor may further improve the TC fetch unit performance. Using a set-associative target buffer with a sensible replacement policy may improve overall benchmark performance.

The fact that the TC allows a smaller pipeline because of the realignment logic required in the MBP permits a fewer number of ID to EX stages. This is an important IPC

performance booster for the TC. Our implementation uses a singly ported instruction cache that allows for simple realignment logic and the performance improvement from reduced ID to EX stages exemplifies this. Future trace cache designs may consider a single L1 instruction cache port so as to minimize the number of pipeline stages.

The trace cache fetch unit design used has three prediction structures: the NTP primary predictor, the NTP secondary predictor and the single branch predictor. The NTP and target buffer both are able to provide CTI targets. Instruction and trace level sequencing are both able to fetch instructions causing a competition between the instruction and traces caches to provide instructions. The trace cache and instruction cache redundantly store instructions. All of these features must come together in the trace cache fetch unit design to fetch instructions efficiently. We see from the single branch predictor and instruction cache plots for the TC that there is a noticeable amount of noise. This fact may indicate that more work can be done to "harmonize" the complex array of features in the TC fetch unit design and possibly reduce its complexity. Doing so may further improve system performance and increase sensitivity to varying structure sizes in the NTP and TC.

8.3 Comparative Analysis

In general, the MBP is more sensitive to data structure size variations than the TC fetch unit design. Our results coincide with Rotenberg et. al who simulate a TC design with a NTP. They discuss the moderate improvement in IPC performance they see as TC fetch unit parameters and sizes are changed. When simulating the trace cache across various benchmarks and various configurations, they see an overall 10% IPC variation. This fact is discussed in the following quote, "Overall performance is not as sensitive to

trace cache size and associativity as one might expect … IPC varies no more than 10 percent over a wide range of configurations" [RBS99].

Studying the actual number of pipeline stages required for each design is a useful area of future research. It may help the MBP to reduce the number of pipeline stages as well by implementing a fast decode latch realignment scheme such as the collapsing buffer [CMMP95].

Further studies of the trade-offs between the performance gained by multi-ported instruction cache designs and those that require only a single port are warranted. A future study may compare the benefit of a shorter pipeline with the number of ports is an example. These studies would require a quantitative analysis of the effect of realignment logic on pipeline depth. They would also require further research to quantify the chip area used by implementing multi-ported caches other than simply the table size.

Issue width is an important parameter affecting the performance of both fetch unit designs across the benchmarks. It improves the performance of certain benchmarks more than others. Both the MBP and TC simulations are affected by an increase in issue width. However, the TC benchmarks' IPC performances are more noticeably improved – particularly bzip2, gzip and TeX.

The IPC performance gained by increasing the L1 instruction cache size is different for the TC and MBP. The MBP is less affected by a very small IC size and its improvement levels off sooner than the TC. For the TC, a width of thirteen provides the best trade-off of performance and chip area. The MBP does well with an eight times smaller L1 IC width of ten.

The IPC performance of the integer benchmarks is significantly more improved by changes in the fetch unit parameters studied than the floating point simulation. This

justifies the fact that more integer benchmarks were studied than floating point simulations.

In general, more studies are needed in the computer architecture field that compare advanced designs using full timing based simulations. Rigorous understanding of the amount of chip area used by specific design categories is very useful for comparative analyses. An industry standard table of basic design element sizes, such as modern instruction caches, would be very helpful. Also, a standard analytical method of describing latency for software simulations, such as the ASIC designer's back annotation for RTL simulations, would provide another level of detail. Our research has shown that there is benefit in investing research in this sort of analysis.

Overall, the MBP and TC designs studied in this work are both effective fetch unit designs for wide dynamically scheduled superscalar processors. Both designs are able to take advantage of increases in issue width so as to fetch more instructions per cycle into the ROB to increasingly take advantage of instruction level parallelism. They each have their unique configurations that result in the best performance.

REFERENCES

[CMMP95] T. Conte, K. Menezes, P. Mills, and B. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates", *22nd International Symposium on Computer Architecture*, June 1995, pp. 333-344.

[DF95] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors", *International Symposium on Microarchitectures*, December 1995, pp. 258 – 263.

[EM98] A. Eden and T. Mudge, "The YAGS Branch Prediction Scheme", *International Symposium on Microarchitecture*, December 1998, pp. 69 – 77.

[HP03] J. Hennessy and D. Patterson, "Computer Architecture A Quantitative Approach", *Morgan Kaufmann Publishers (ISBN: 1-55860-724-2)*, 2003.

[JRS97] Q. Jacobson, E. Rotenberg and J. E. Smith, "Path-Based Next Trace Prediction," *Proc. 30th Int'l Symp. High Performance Microarchitecture*, December 1997, pp. 14 - 23.

[K02] D. Koppelman, "The Benefit of Multiple Branch Prediction on Dynamically Scheduled Systems," *Workshop on Duplicating, Deconstructing, and Debunking Held in conjunction with the 29th International Symposium on Computer Architecture*, May 2002, pp. 42 – 51.

[LS84] J. Lee and A Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *Computer*, 17(1), January 1984.

[M93] S. McFarling, "Combining Branch Predictors," *WRL Technical Note TN-36*, June 1993.

[MH86] S. McFarling and J. Hennesy, "Reducing the Cost of Branches", *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 396-403.

[MSP88] S. Melvin and M. Shebanow, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines", *Proceedings of the 21st Annual International Workshop on Microprogramming and Microarchitecture*, November 1988.

[PSR92] S. Pan, K. So and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", *Proceedings of ASPLOV V*, October 1992, pp. 76 – 84.

[PTM98] M. Postiff, G. Tyson and T. Mudge, "Performance Limits of Trace Caches", *Journal of Instruction-Level Parallelism*, 1998.

[RBS96] E. Rotenberg, S. Bennett and J. E. Smith, "Trace Cache: A Low Latency Approach to High bandwidth Instruction Fetching," *Proceedings if the 30th International Symposium of High Performance Microarchitectures*, December 1997, pp. 138 – 148.

[RBS99] E. Rotenberg, S. Bennett and J. E. Smith, "A Trace Cache Microarchitecture and Evaluation," *IEEE Transaction on Computers*, vol. 48, no. 2, February 1999, pp. 111 - 120.

[RJSS97] E. Rotenberg, Q. Jacobson, Y. Sazeides and J. Smith, " Trace Processors", *Proceedings if the 30th International Symposium of High Performance Microarchitectures*, December, 1997.

[S81] J. Smith, "A Study of Branch Prediction Strategies", *Proceedings if the 8th International Symposium of Computer Architecture*, May 1981, pp. 135 – 138.

[UI04] "The RSIM Project", *Department of Computer Science University of Illinois at Urbana-Champaign*, 2004, http://rsim.cs.uiuc.edu/.

[VRBV02] H. Vandierendonck, A. Ramirez, K. Bosschere and M. Valero, "A Comparative Study of Redundancy in Trace Caches", *Proceedings of the International EuroPat Conference*, August 2002.

[WB97] S. Wallace and N. Bagherzadeh, "Multiple Branch and Block Prediction", *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, 1997, pp. 94 – 104.

[SJSM96] A. Seznec, S. Jourdan, P. Sainrat and P. Michaud, "Multiple Block Ahead Branch Predictor", *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[WG94] D. Weaver and T. Germond, "The SPARC Architecture manual Version 9", *PTR Prentice Hall, Prentice-Hall Inc. (ISBN: 0-13-825001-4)*, 1994.

[YMP93] T. Yeh, D. Marr, Y. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache", *in Proceedings of the International Conference on Supercomputing*, 1993, pp. 67 – 76.

[YP92] T. Yeh and Y. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction", *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 124 – 134.

[YP93] T. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors That Use Two-Levels of Branch History", *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993, pp. 257 – 266.

APPENDIX: GLOSSARY

BAC

Branch Address Cache. A cache that stores branches and their target. It is used to construct a path through multiple basic blocks to facilitate the fetching multiple basic blocks per cycle.

Basic Block

A sequence of instructions in run-time order that contain a single control transfer instruction at the end (there may be a delay slot following the CTI). The instructions have sequential program counter values. The basic block has only one exit point and might have more than one entry point.

CTI

Control Transfer Instruction. An instruction that can alter future program counter values in the run-time instruction stream.

Delay Slot

The instruction immediately following many branches in the SPARC architecture. The delay slot is the next sequential instruction after the branch and following the delay slot is the branch's target.

Dynamic Scheduling

Dynamic scheduling is a processor scheduling technique that allows out-of-order execution of instructions. This is done to minimize stalls due to data hazards.

IPC

Instructions Per Cycle. This is the ratio of the total number of instructions committed from the reorder buffer with respect to the total number of cycles.

MBP

Multiple Branch Predictor. A fetch unit architecture that incorporates a multiple branch predictor to predict and thereby issue more than a single basic block per cycle.

NTP

Next Trace Predictor. The next trace predictor is a trace level predictor used to select the next trace to issue.

PC

Program Counter. The address into program memory for this instruction.

Reorder Buffer

The acronym for Reorder Buffer is *ROB*. The ROB is a structure in the processor that holds instructions in program order. It is used, on rare occasions, to set the processor state to what it would be if every instruction up to a particular instruction executed and no instruction at or after the particular instruction executed. This is done if the particular instruction raised an exception, and for other reasons.

TC

Trace Cache. The fetch unit architecture that uses a trace cache to hold instructions in their committed order so that they can be reissued under the direction of some prediction mechanism. It can easily provide instructions from non-contiguous regions of the program.

Trace

A trace is a fixed length record of instructions in their committed order. It can be composed of more than one basic block. Traces are stored in a trace cache.

Trace ID

Trace Identifier. This is the identifier used to uniquely identify a trace.

TSB

Trace Search Back. A trace cache feature introduced here in which a predicted instruction can be anywhere in a trace, not just first, vastly increasing TC efficiency.

Width

The log base two of the number of table entries or number of cache lines

APPENDIX: RESULTS

Figure 9. MBP Issue Width, MBP Order 1

Figure 10. MBP Issue Width, MBP Order 2

Figure 11. MBP Issue Width, MBP Order 3

Figure 12. MBP Issue Width, MBP Order 4

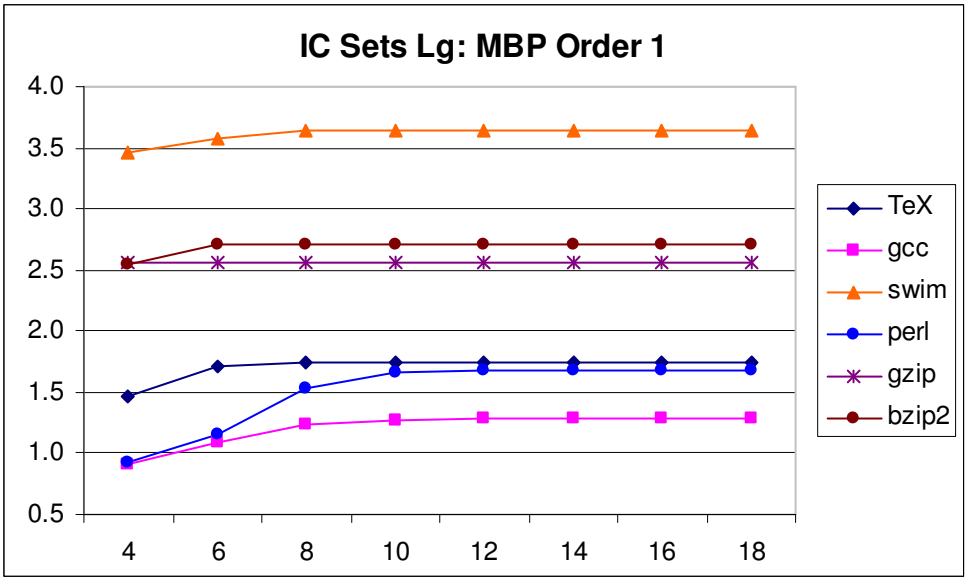

Figure 13. MBP IC Sets Lg, MBP Order 1

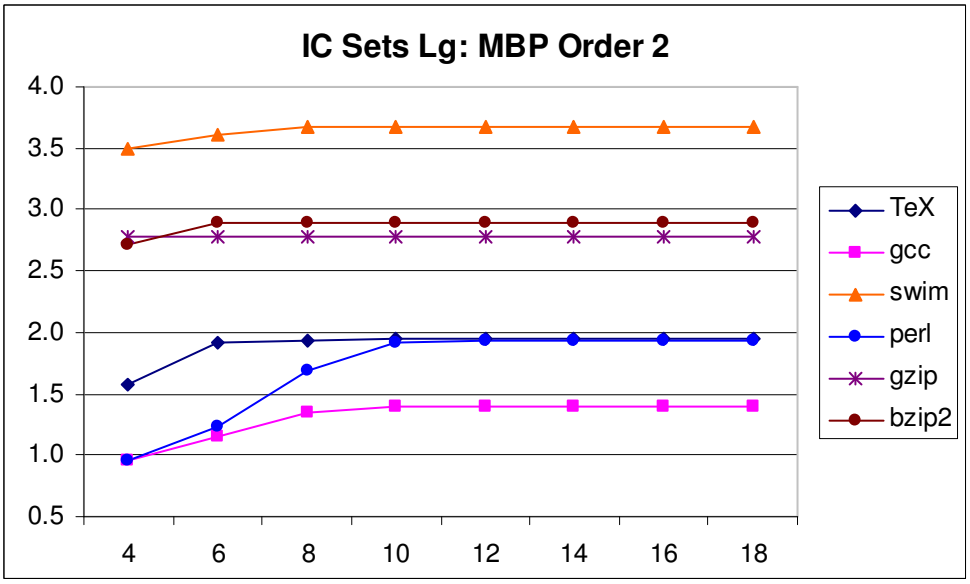

Figure 14. MBP IC Sets Lg, MBP Order 2

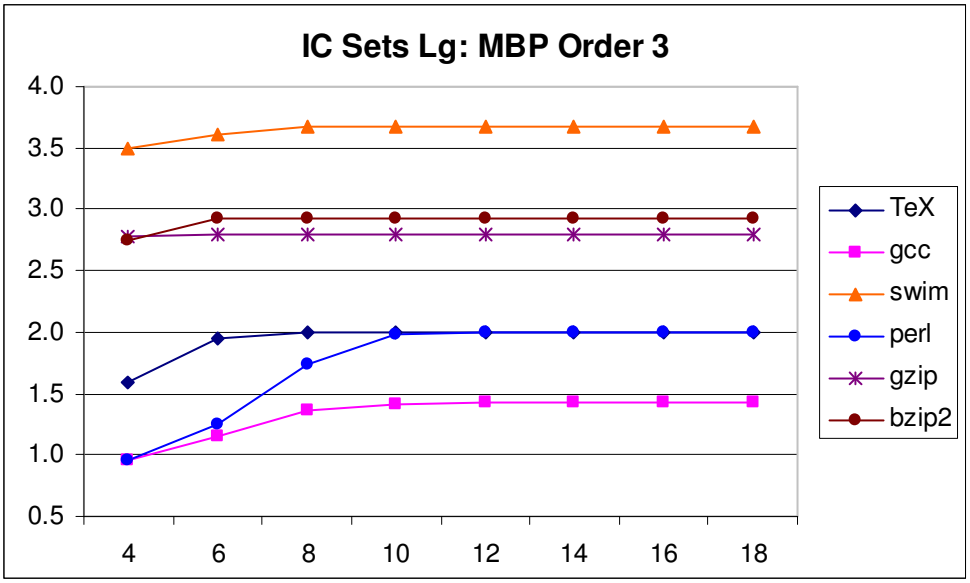

Figure 15. MBP IC Sets Lg, MBP Order 3

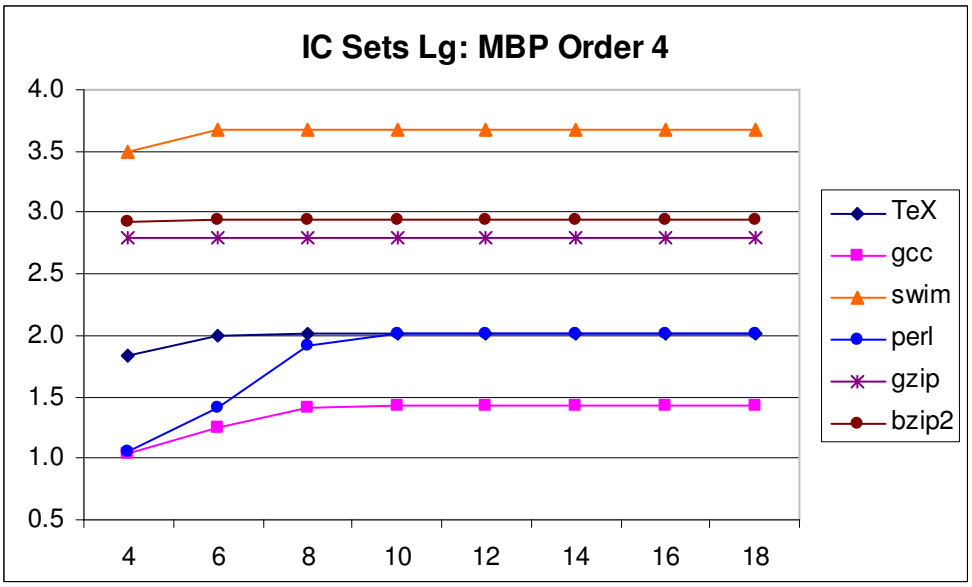

Figure 16. MBP IC Sets Lg, MBP Order 4

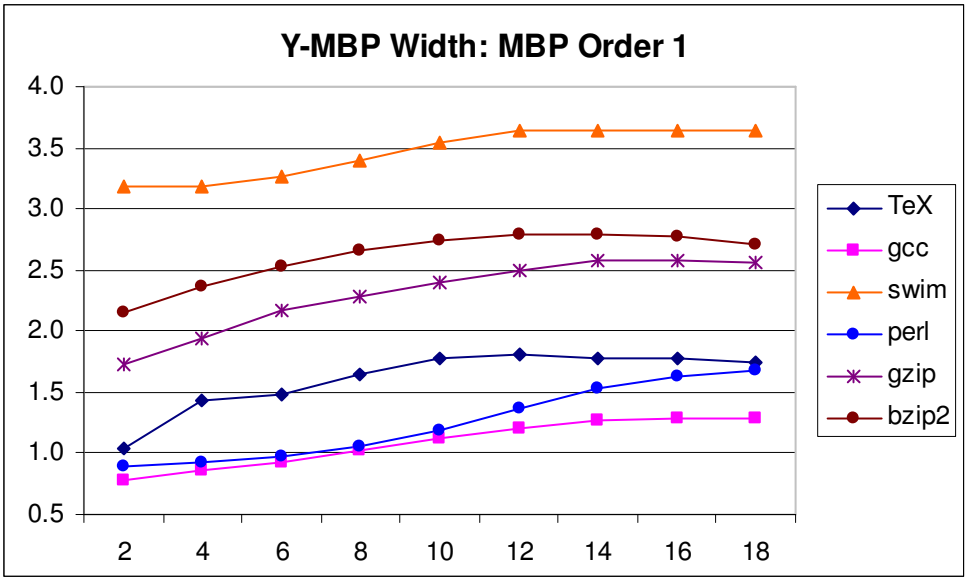

Figure 17. MBP Y-MBP Width, MBP Order 1

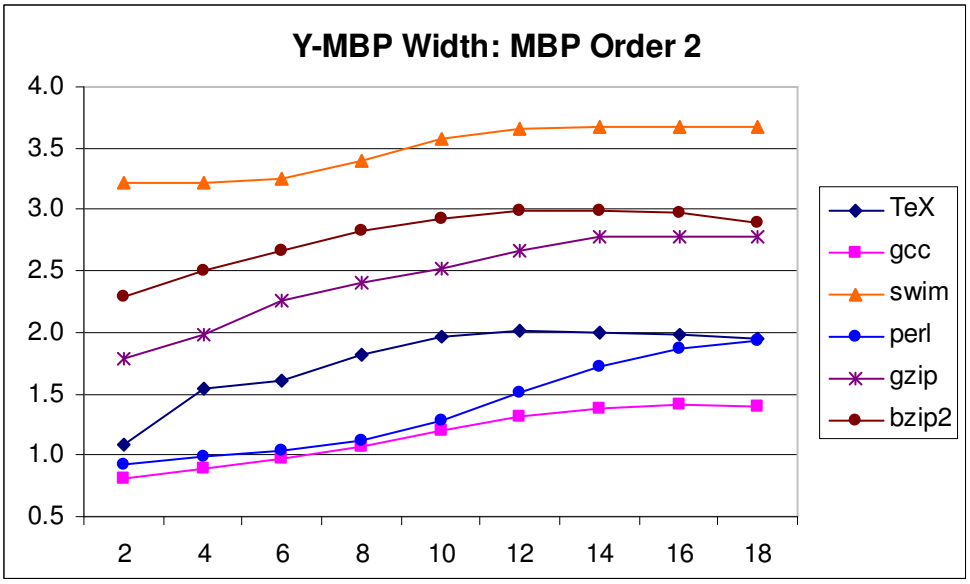

Figure 18. MBP Y-MBP Width, MBP Order 2

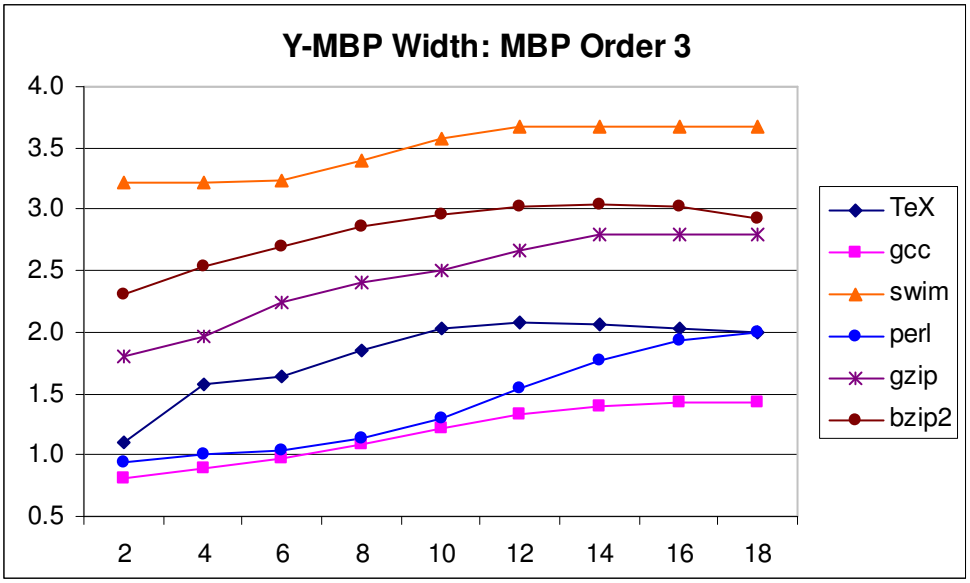

Figure 19. MBP Y-MBP Width, MBP Order 3

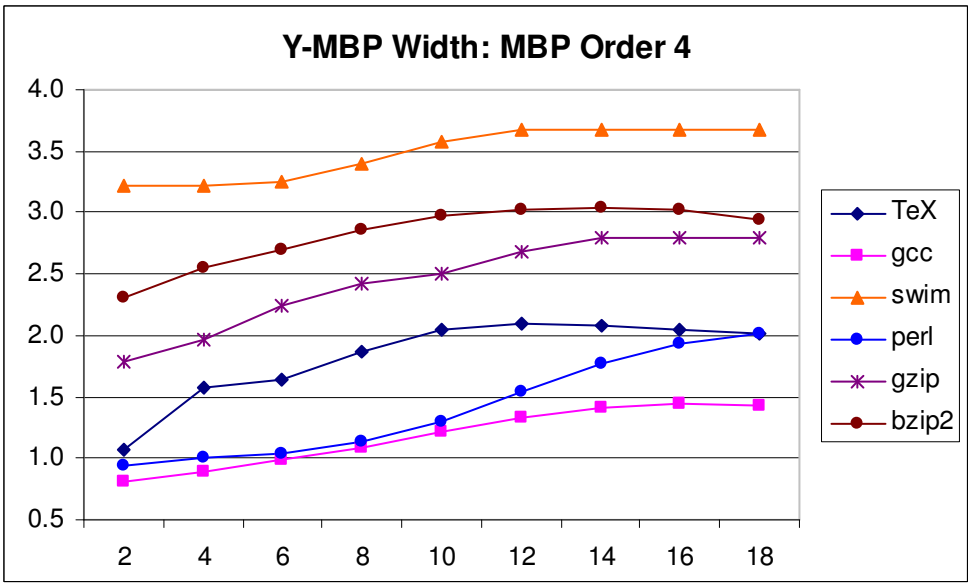

Figure 20. MBP Y-MBP Width, MBP Order 4

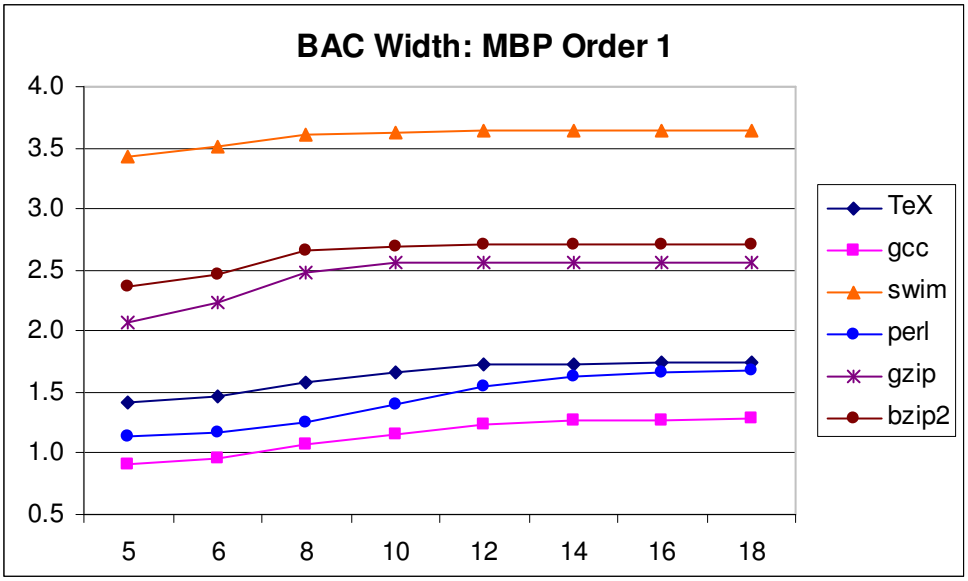

Figure 21. BAC Width, MBP Order 1

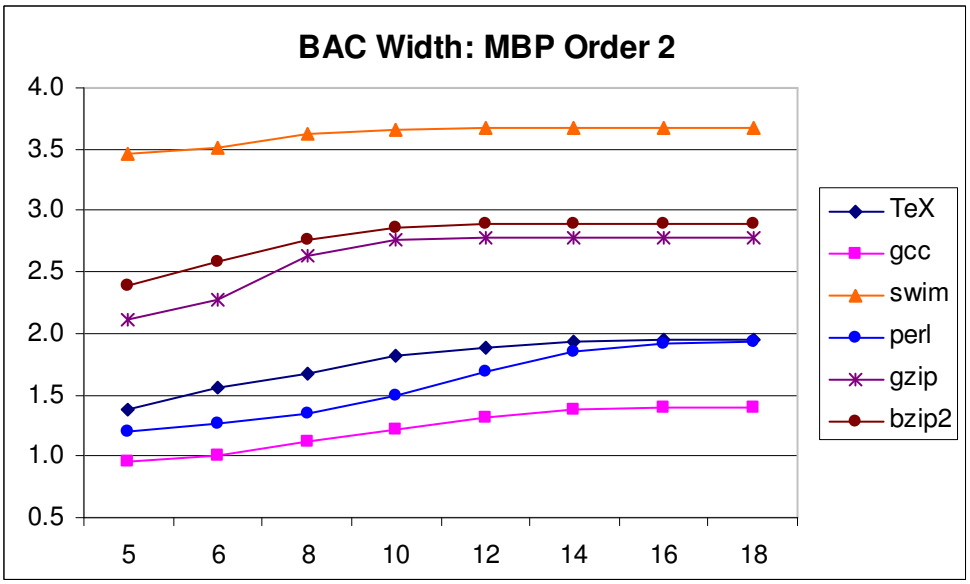

Figure 22. BAC Width, MBP Order 2

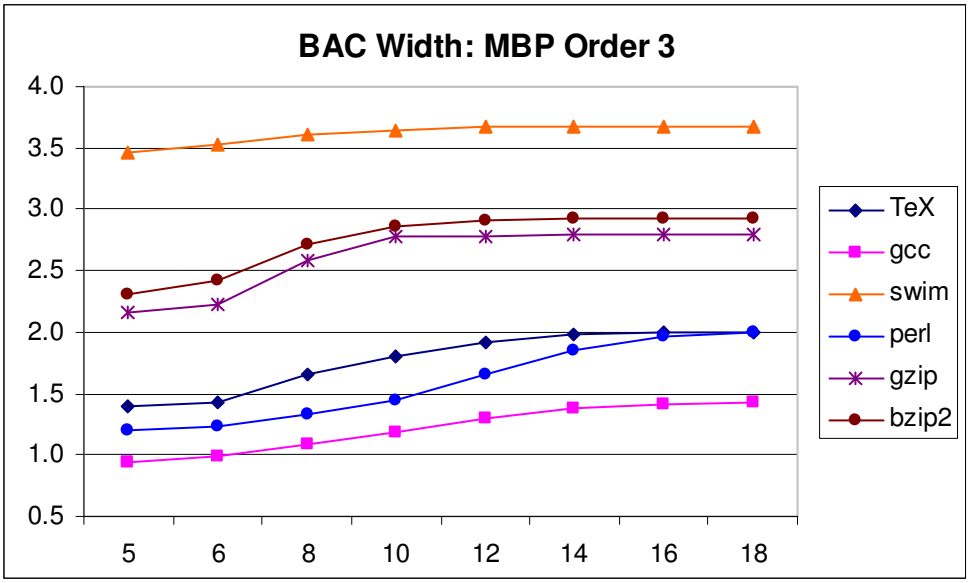

Figure 23. BAC Width, MBP Order 3

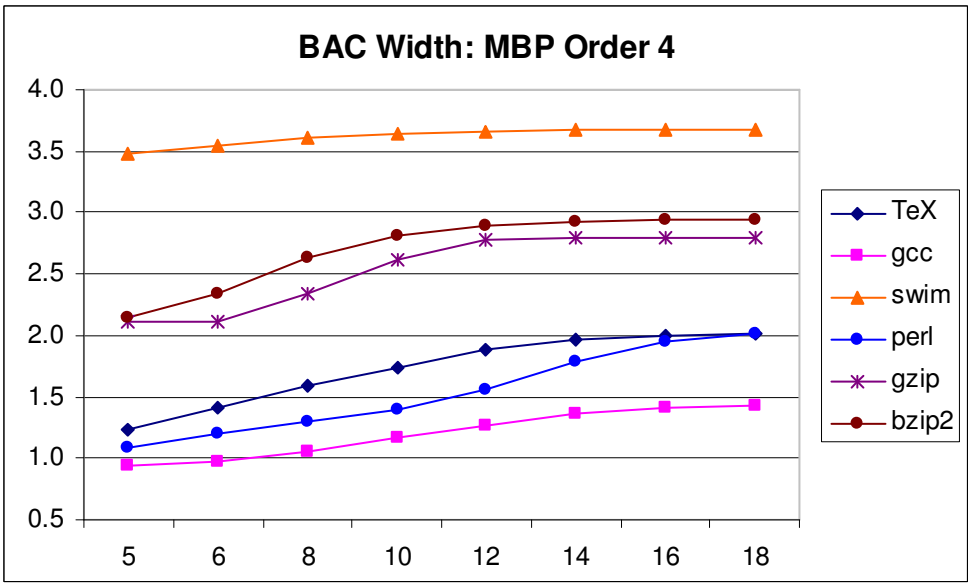

Figure 24. BAC Width, MBP Order 4

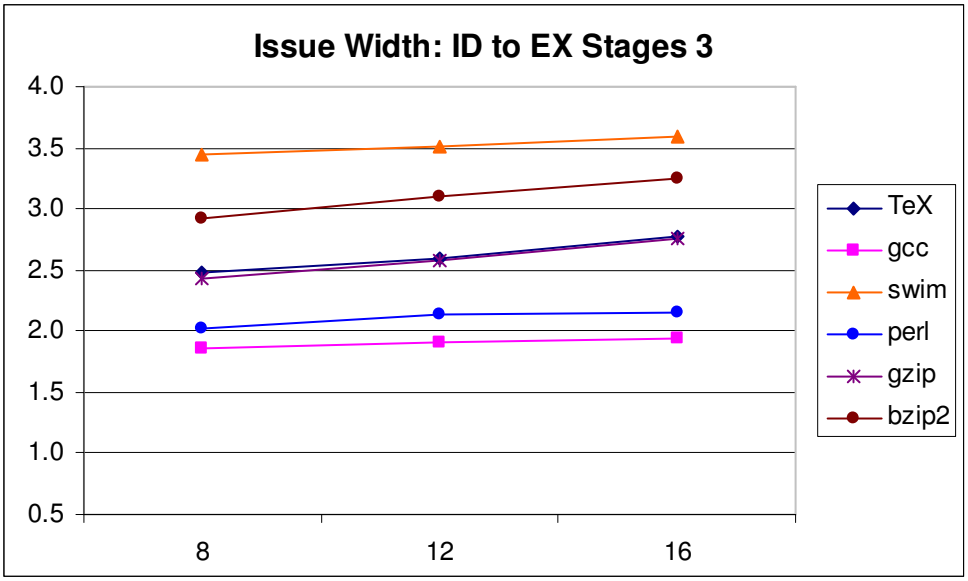

Figure 25. TC Issue Width, ID to EX Stages 3

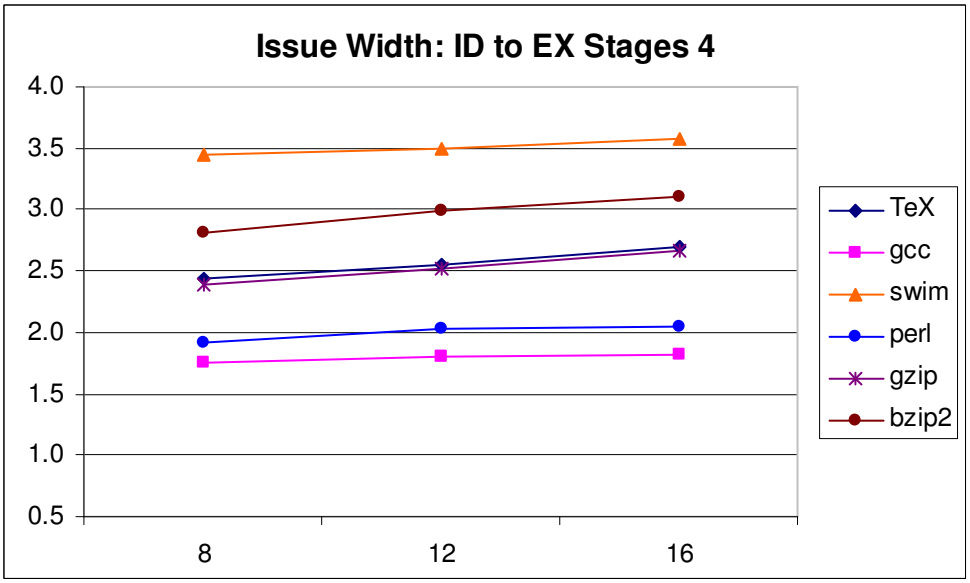

Figure 26. TC Issue Width, ID to EX Stages 4

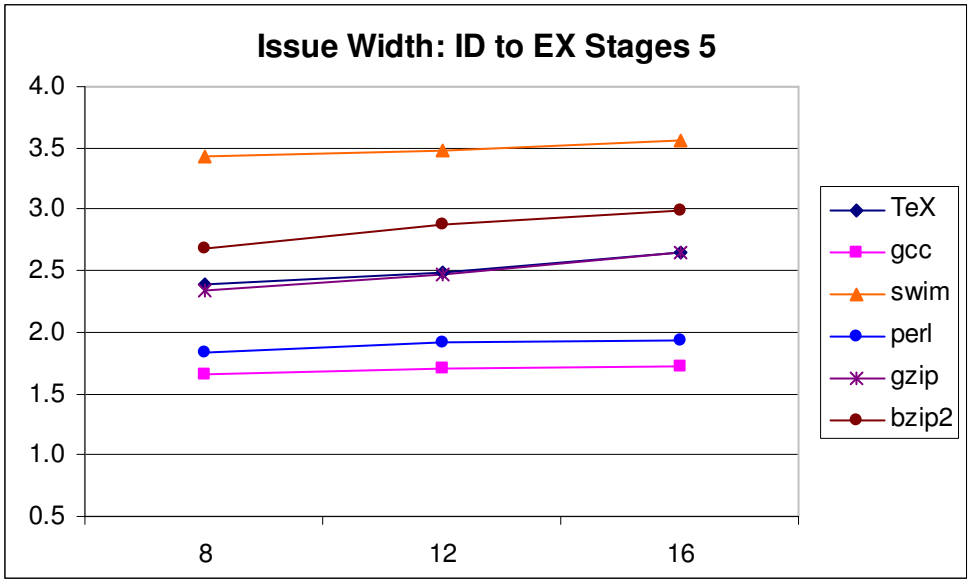

Figure 27. TC Issue Width, ID to EX Stages 5

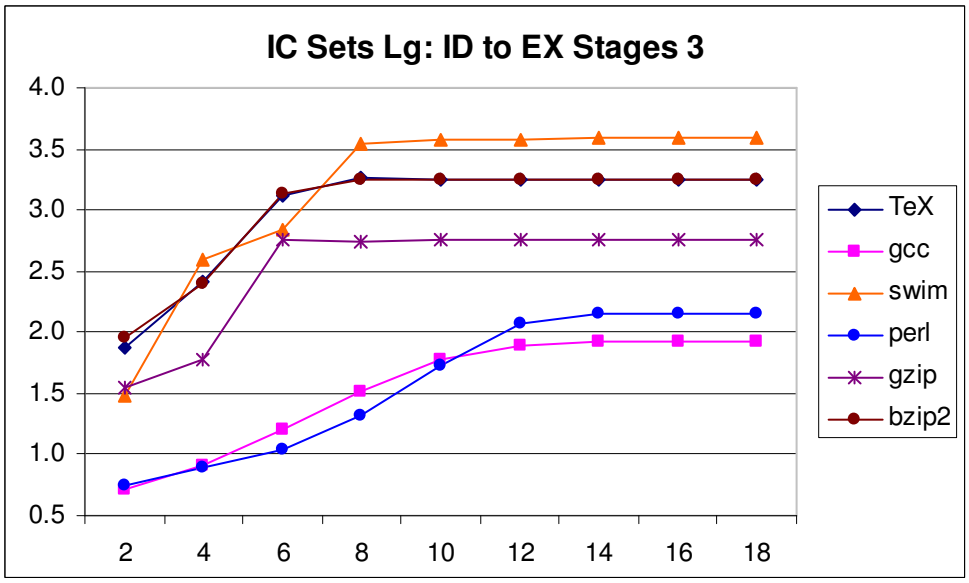

Figure 28. TC IC Sets Lg, ID to EX Stages 3

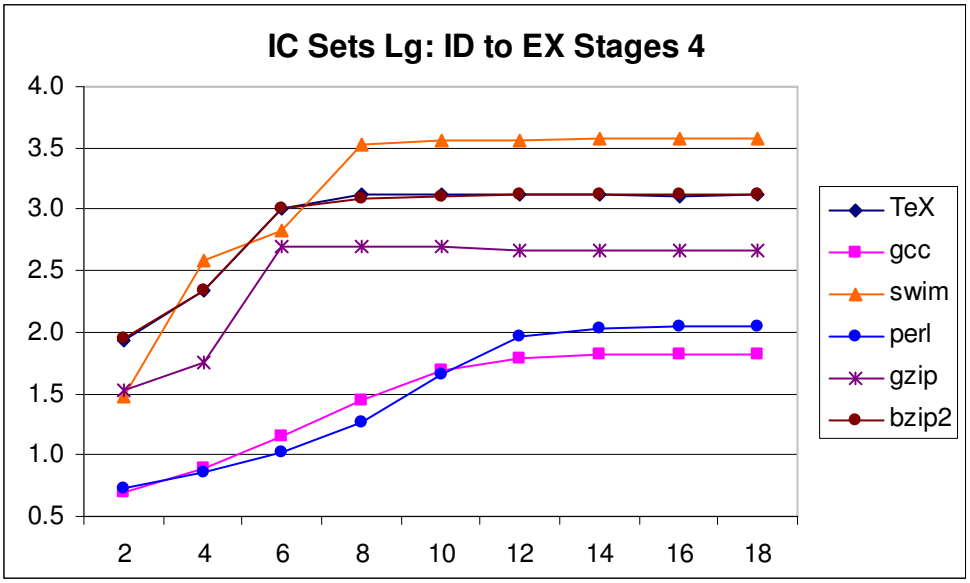

Figure 29. TC IC Sets Lg, ID to EX Stages 4

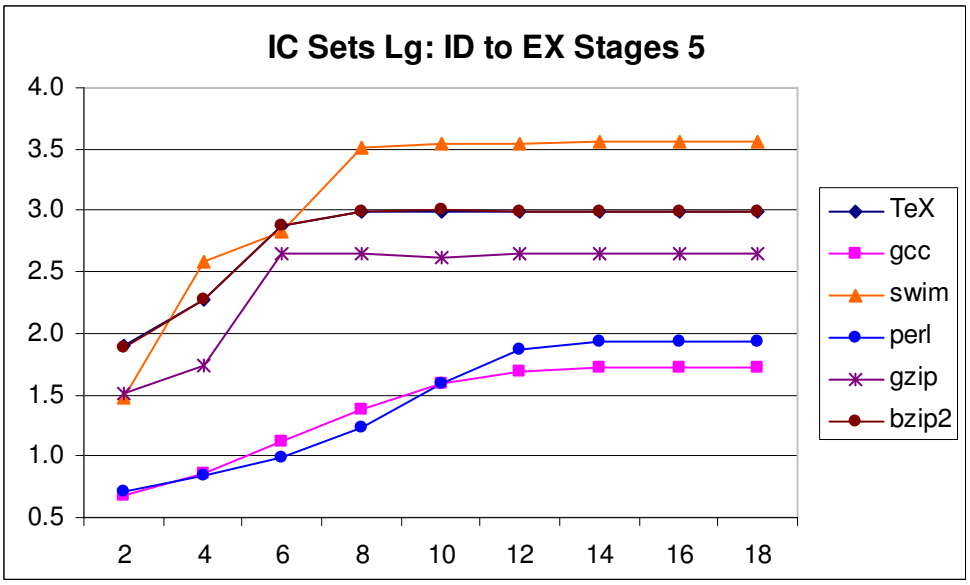

Figure 30. TC IC Sets Lg, ID to EX Stages 5

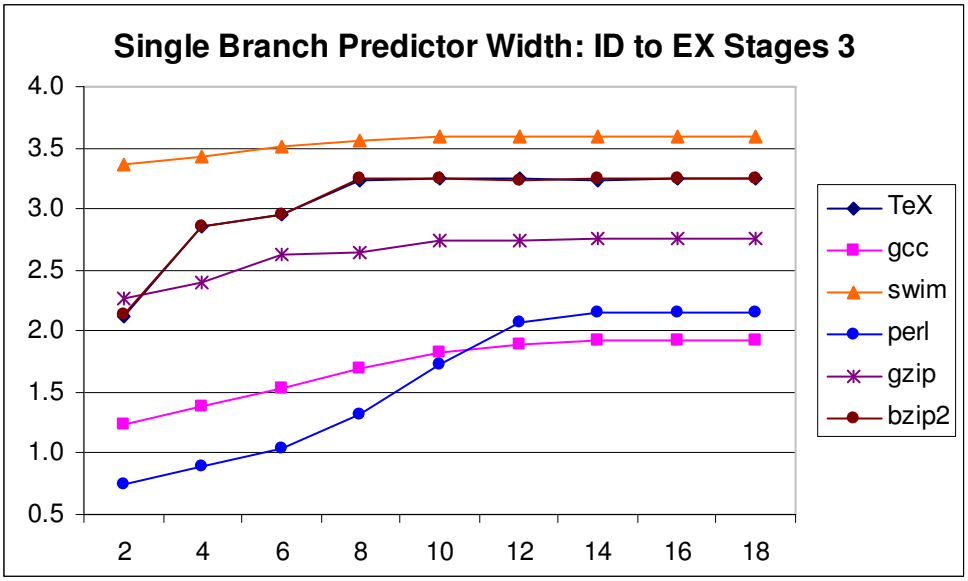

Figure 31. TC Single Branch Predictor Width, ID to EX Stages 3

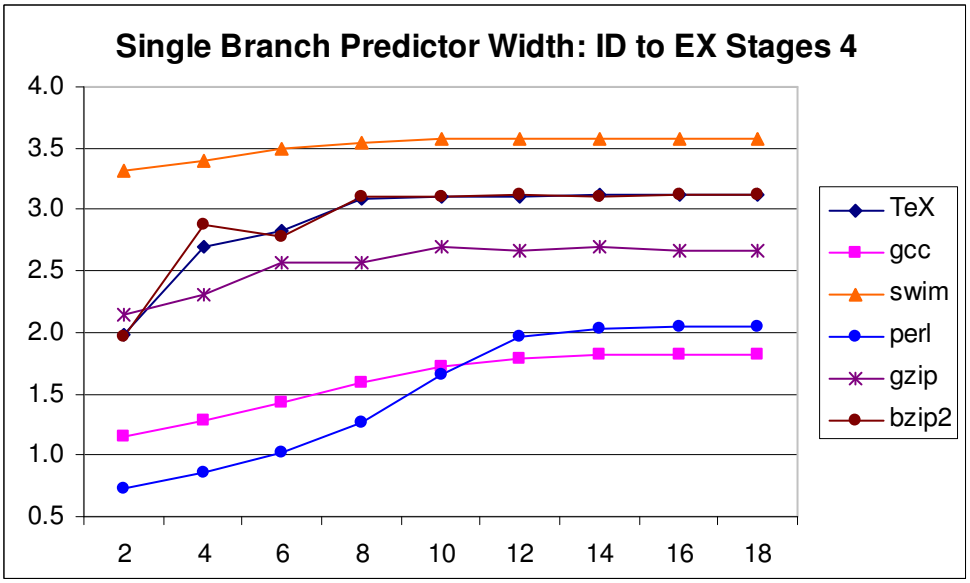

Figure 32. TC Single Branch Predictor Width, ID to EX Stages 4

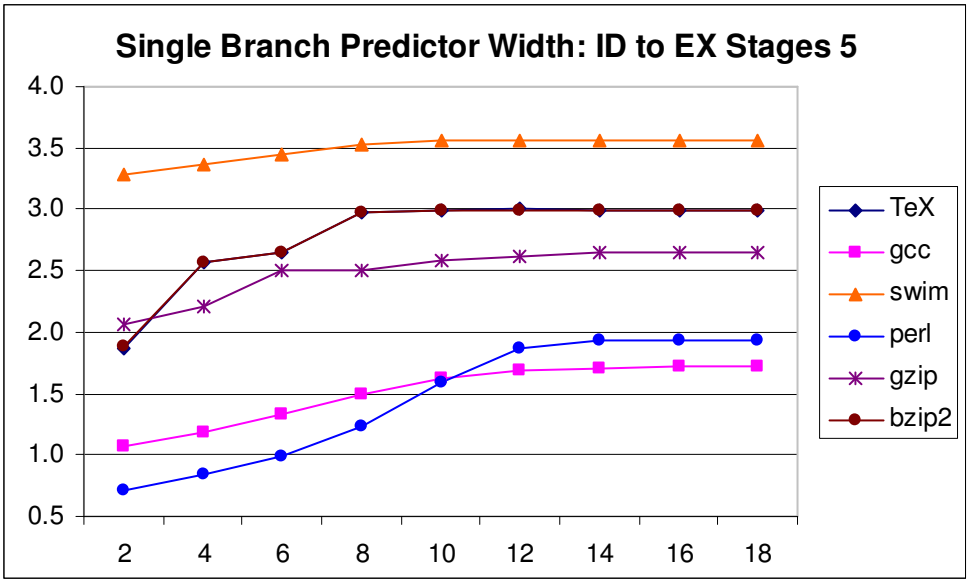

Figure 33. TC Single Branch Predictor Width, ID to EX Stages 5

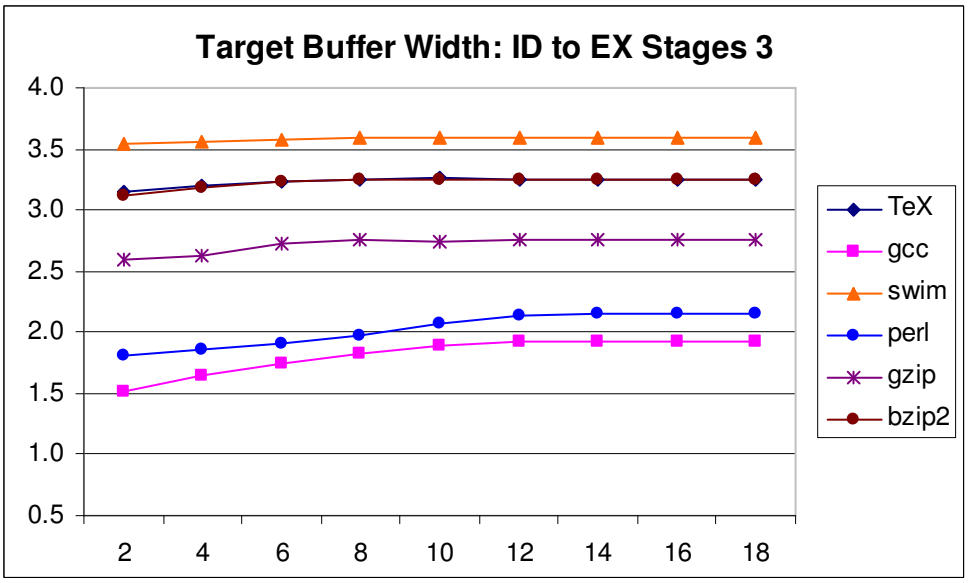

Figure 34. TC Target Buffer Width, ID to EX Stages 3

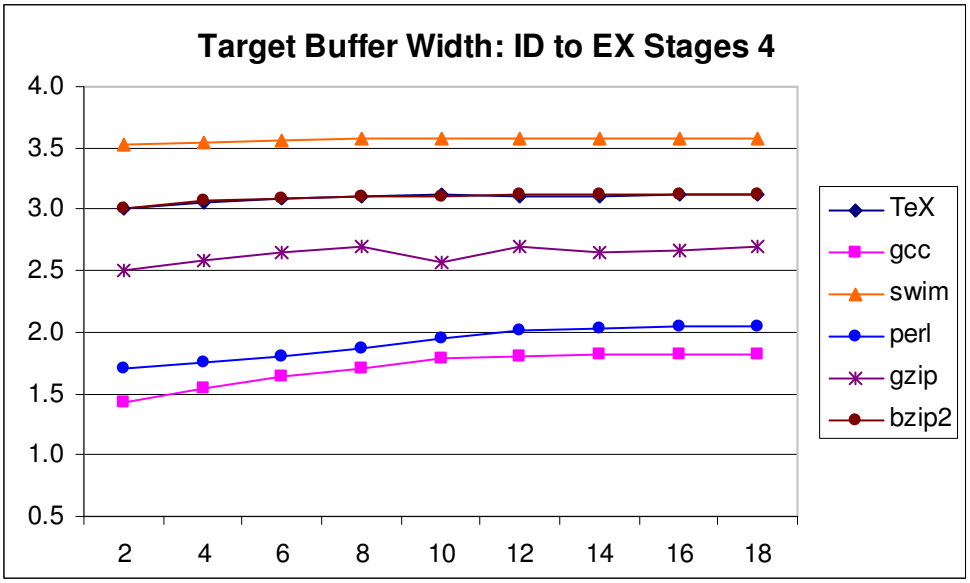

Figure 35. TC Target Buffer Width, ID to EX Stages 4

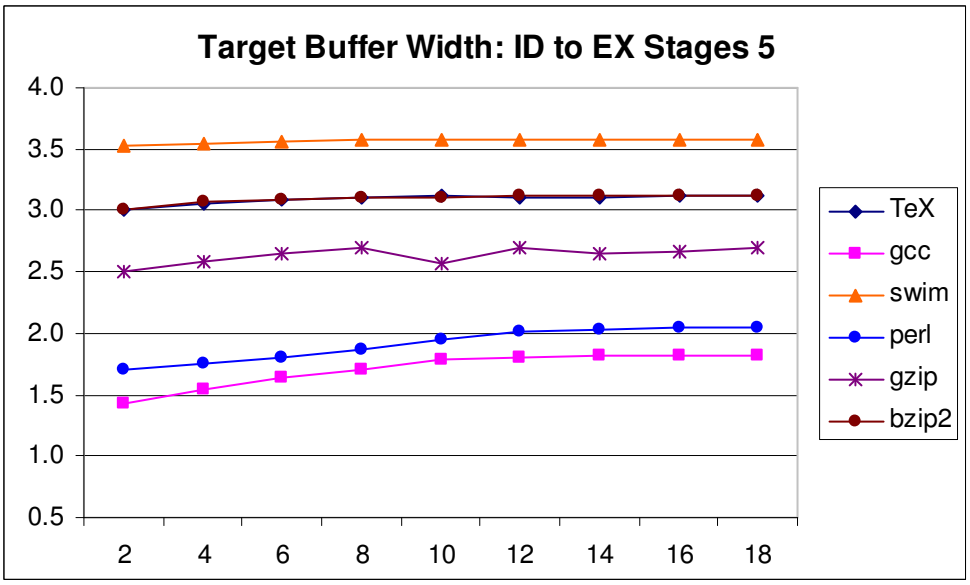

Figure 36. TC Target Buffer Width, ID to EX Stages 5

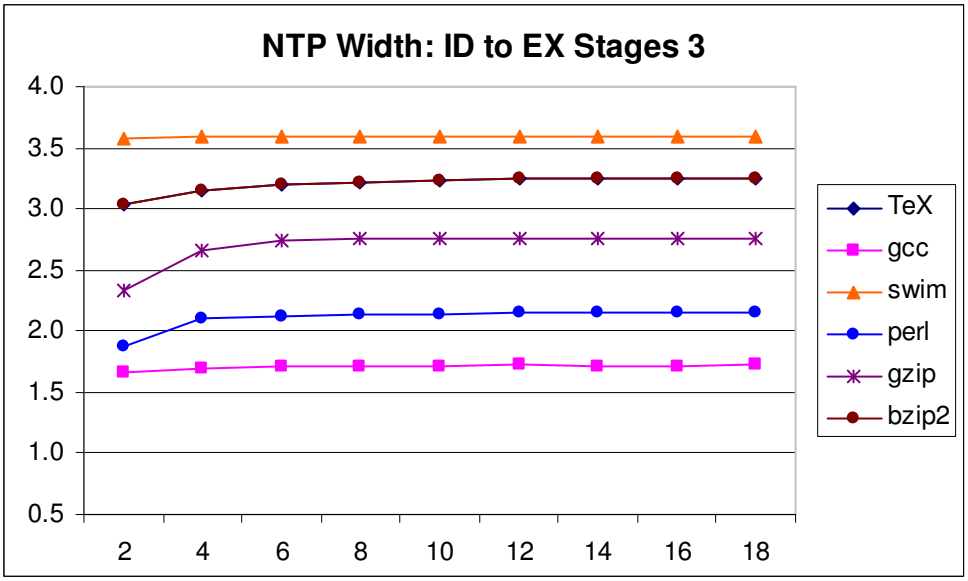

Figure 37. NTP Width, ID to EX Stages 3

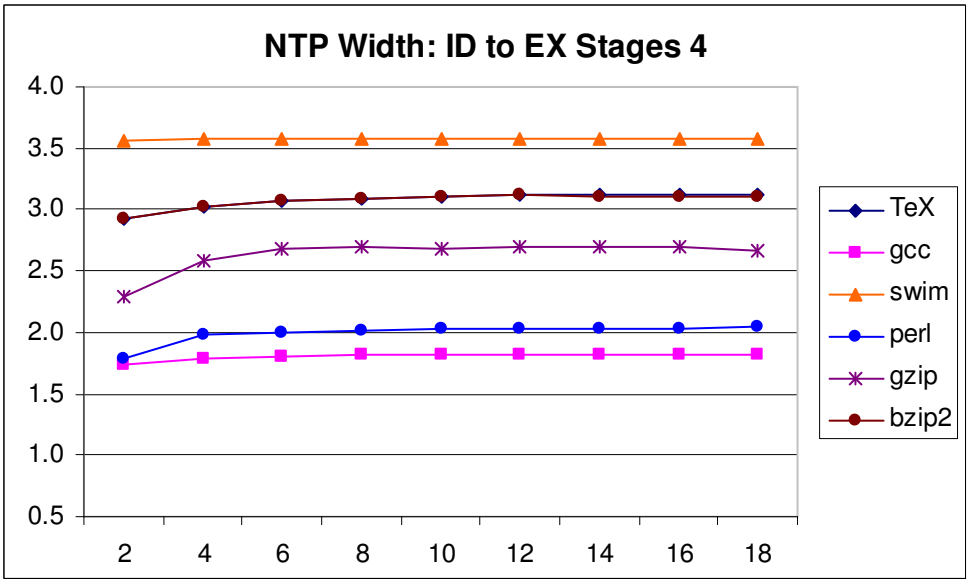

Figure 38. NTP Width, ID to EX Stages 4

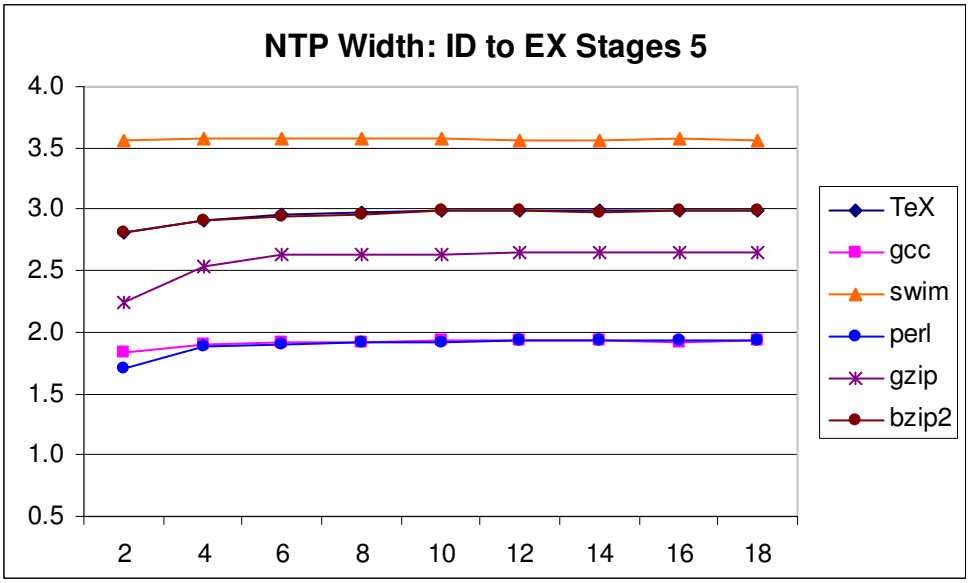

Figure 39. NTP Width, ID to EX Stages 5

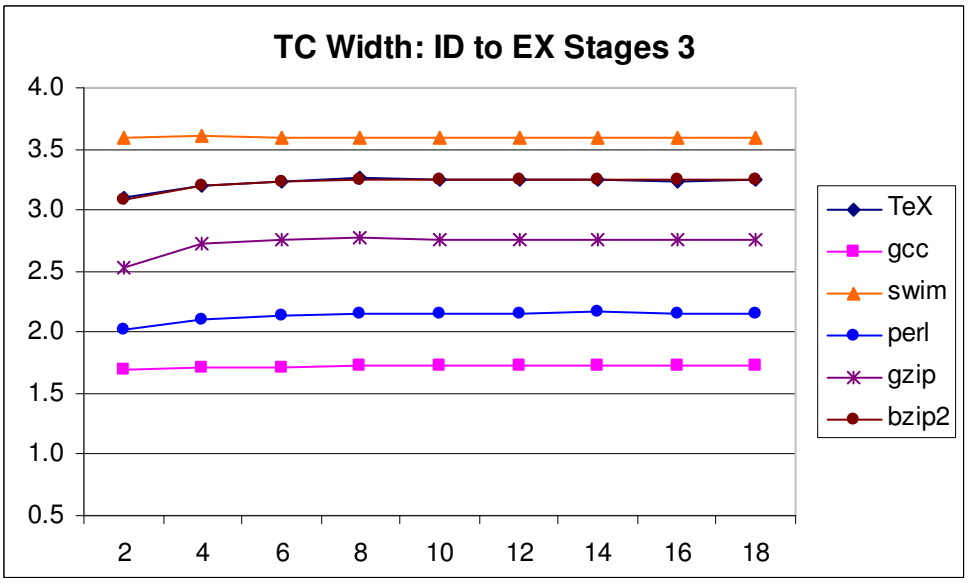

Figure 40. Trace Cache Width, ID to EX Stages 3

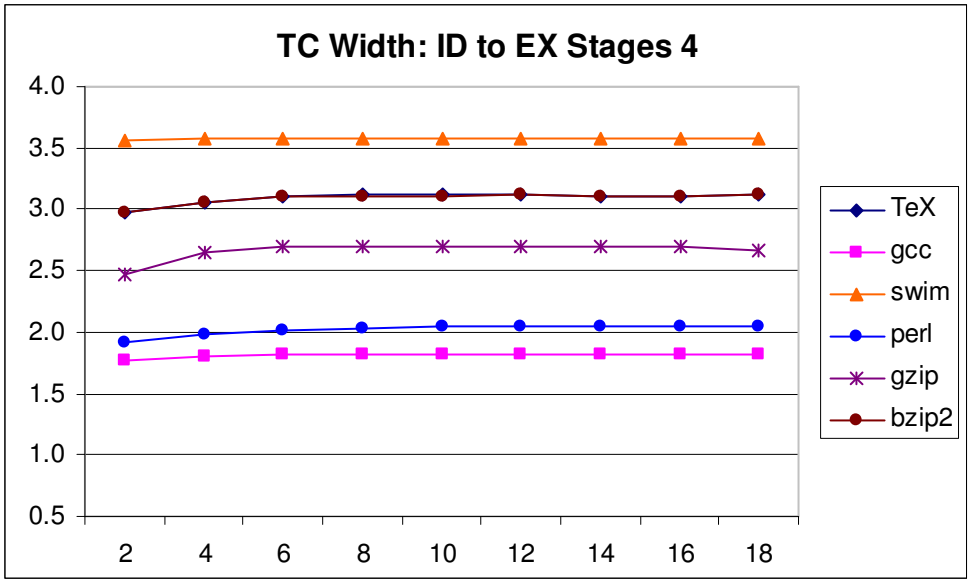

Figure 41. Trace Cache Width, ID to EX Stages 4

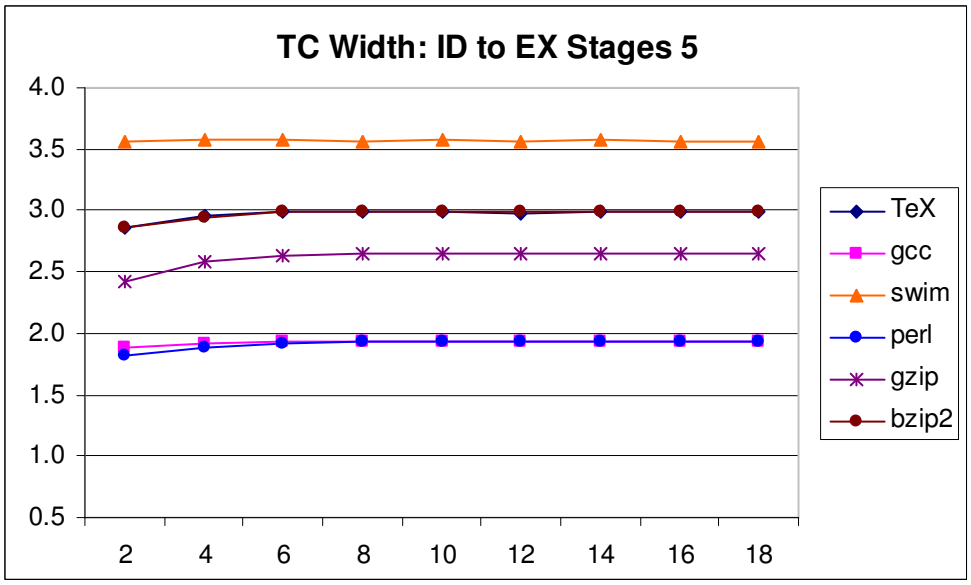

Figure 42. Trace Cache Width, ID to EX Stages 5

VITA

Slade Maurer was born in 1977 in Portland, Maine. He finished his undergraduate studies in computer science at the Southeastern Louisiana University in December 1998. In August 1999 he came to Louisiana State University to pursue graduate studies in electrical and computer engineering. Currently he is a candidate for the degree of Master of Science in Electrical Engineering, which will be awarded in December 2004.