Design of Trace Caches for High Bandwidth Instruction Fetching

by

Michael Sung

S.B., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1997

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

May 1998

Author_____
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by_____
Arvind
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Design of Trace Caches for High Bandwidth Instruction Fetching

by

Michael Sung

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 1998 in partial fulfillment of the requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

In modern high performance microprocessors, there has been a trend toward increased superscalarity and deeper speculation to extract instruction level parallelism. As issue rates rise, more aggressive instruction fetch mechanisms are needed to be able to fetch multiple basic blocks in a given cycle. One such fetch mechanism that shows a great deal of promise is the trace cache, originally proposed by Rotenburg, et. al. In this thesis, critical design issues regarding the trace cache fetch mechanism are explored in order to develop techniques to further improve trace cache performance. The thesis research presents an optimized trace cache design that show an average 34.9% improvement for integer benchmarks and 11.0% improvement for floating-point benchmarks, relative to the originally proposed trace cache design. This corresponds to a 67.9% and 16.3% improvement in fetch bandwidth over a traditional instruction cache, for integer and floating-point benchmarks respectively. The results demonstrate the viability of the trace cache as a high performance fetch mechanism and provide justification for additional research .

Thesis Supervisor: Arvind
Title: Professor of Computer Science and Engineering

Acknowledgments

First and foremost, I would like to express my gratitude to my Mom and Dad, for all their love and encouragement throughout the years. I definitely would not be where I am today, had it not been for them. I also want to thank my little sister, Emily, who has always made me feel important and special.

To Marilyn, I give my utmost appreciation and thankfulness. I wish I could show her how grateful I am for her unconditional support. She has been an inspiration to me, patiently sticking by my side through the countless days and nights while I worked on my thesis.

A special thanks goes to Marty Deneroff, who has been the most incredible resource while I have been at Silicon Graphics. I would also like to acknowledge Professor Arvind for his guidance and helpful suggestions.

I must also recognize Silicon Graphics, Inc. for all the resources and opportunities that were provided to me. The company has been instrumental to my education and growth, and I am indebted to their support.

Finally, I would like to pay a tribute to MIT, which has been my home for the last five years. They have been the most memorable in my life, and I will always reserve a special place in my heart for those that I have met and befriended while at the 'Tute.

Table of Contents

List of Figures and Tables

Figures

Tables

Chapter 1

Introduction

1.1 Background and Motivation

In today's modern microprocessor, many aggressive techniques are employed in order to exploit instruction level parallelism. The trend of these high performance superscalar microprocessors has been to widen dispatch and issue paths in order to increase their effective IPC (instructions per cycle). To this end, each successive microprocessor generation has increased parallel functional units, more physical registers, larger instruction issue windows, and deeper branch speculation. Currently, microprocessors such as the MIPS R10000, Sun UltraSPARC, and AMD K5 are capable of issuing up to four instructions per cycle [28], [29], [30]. Next generation microprocessors will have even higher issue rates as hardware parallelism increases.

The focus on increasing instruction issue width in recent years has introduced a possible performance bottleneck in these superscalar architectures. In order to fully exploit instruction level parallelism, it is necessary to balance the increased issue rate of these superscalar processors with sufficient instruction fetch bandwidth. If instructions cannot be issued fast enough, then overall performance will drop as execution resources lie idle. Thus, there is a need to explore new techniques for increasing instruction fetch bandwidth. Increasing fetch bandwidth can improve overall performance in two ways. First and foremost, the increased number of fetched instructions can be used to fill idle functional unit resources. In addition, since there are more instructions, more independent instructions can be found for issue.

Traditional superscalar architectures can be thought as being divided into two main components: an instruction fetch unit and an execution unit. The fetch and execution units are connected by an instruction issue buffer which serves to decouple the two components. The instruction fetch engine fetches and decodes the dynamic sequence of instructions in the program and places them in the

instruction issue buffer. The instruction execution engine takes instructions from the issue buffer and executes them as soon as data dependencies are resolved and resources become available. Control dependencies, in the form of branch and jump instructions, provide a feedback mechanism between the fetch and execution engines.

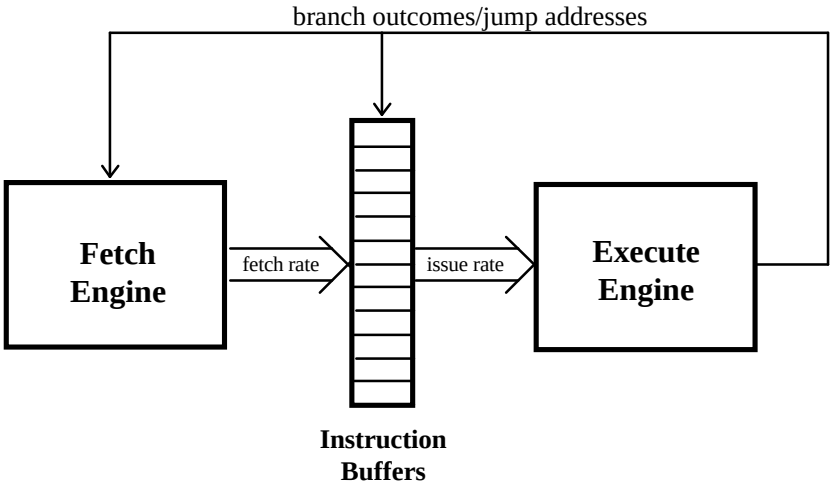

Figure 1-1: Typical Superscalar Design with Fetch and Execute Engines

From Figure 1-1, it is apparent that increasing instruction fetch bandwidth is necessary as the execution engine becomes more parallel and issue rates rise. Instruction fetch bandwidth has traditionally been controlled by factors such as the cache hit rate and branch prediction accuracy. The cache hit rate affects fetch bandwidth because overall instruction cache performance is directly proportional to it. Branch prediction accuracy affects the fetch bandwidth because of the mispredict recovery time. These two factors have long been identified as fetch performance bottlenecks and are relatively well-researched topics. The design of instruction caches has been studied in great detail in order to lessen the impact of instruction cache misses on fetch bandwidth [31]-[36]. Likewise, there have been many studies done to improve branch prediction accuracy [16]-[25].

To date, the techniques developed to reduce instruction cache misses and increase branch prediction accuracy have been very successful in improving fetch bandwidth. However, as the issue rates for superscalar processors increase beyond four, other additional factors become increasingly important to instruction fetch performance. These factors include the frequency of control transfer instructions, branch prediction throughput, noncontiguous instruction alignment, and fetch unit latency. Conventional instruction caches are limited in their effectiveness by control transfer in-

structions present within the instruction stream. Since branch prediction is currently limited to the one branch, an instruction cache can only fetch up to one basic block's worth of instructions per cycle. Since control transfer instructions occur relatively frequently in an instruction stream (approximately 1 in 5 instructions for common integer code), they create a bottleneck in the instruction fetch bandwidth of the processor. Table 1-1 provides the percentage of control transfer instructions in a program and the resulting average basic block sizes for the SPECint95 benchmarks.

SPECint95 Benchmark	% Control Transfer Instructions	Average Basic Block Size
go	15.7%	6.32
m88ksim	16.8%	5.88
gcc	18.2%	5.41
compress	13.7%	7.33
li	21.6%	4.63
ijpeg	6.7%	13.87
perl	18.8%	5.32
vortex	21.7%	4.62

Table 1-1: Average Dynamic Basic Block Sizes for SPECint95 Benchmarks

In order to increase instruction fetch bandwidth, one needs to either increase the basic block size or fetch multiple basic blocks every cycle. A great deal of compiler research has been conducted in order to increase the basic block length, including research on how to decrease the frequency of taken branches [37], [38]. The other possibility is to increase the number of branch predictions, or branch throughput, that can be made during a given cycle. Currently, only one branch can be predicted ahead of time. This limits the amount of instructions that can be fetched, given the frequency of control transfer instructions in a typical fetch stream.

Unfortunately, increasing branch throughput does not solve everything. Even if multiple branch prediction is employed, the presence of taken branches results in noncontiguous instruction fetching. Instruction caches store static instruction blocks from memory. With multiple branch prediction, specialized interleaved instruction caches can be made to fetch beyond a control transfer instruction if the next basic block is contiguous [2]. However, in general dynamic instruction sequences do not always lie in contiguous cache locations, as in the case of taken branches and

jumps. This presents a problem, as a conventional instruction cache cannot fetch noncontiguous blocks in a single cycle. Since taken conditional branches occur rather frequently (estimates of around 70% of the time [41], instruction fetch bandwidth is severely limited. Figure 1-2 illustrates the problem with taken branches and jumps:

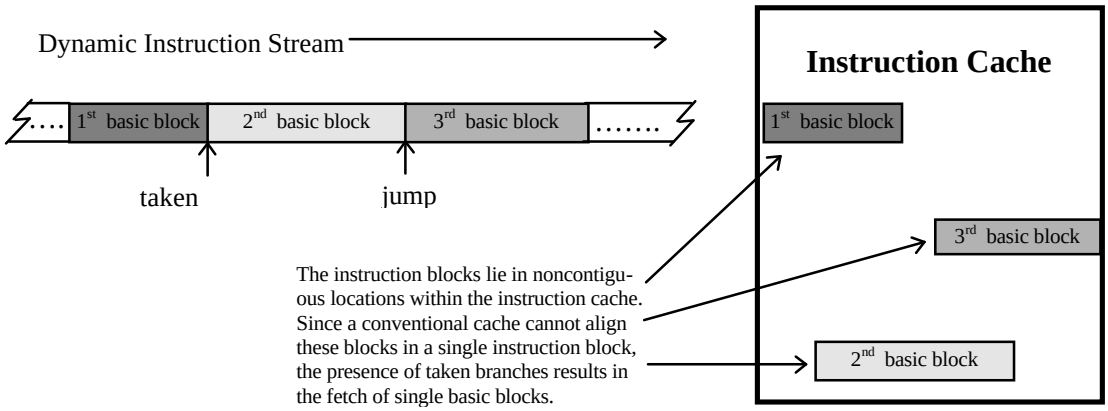

Figure 1-2: Noncontiguous Basic Blocks From Taken Branches

Finally, fetch unit lateny has a large effect on processor performance since incorrect control transfer instruction speculation results in the need to flush the fetch pipeline. Increasing the fetch latency, as is necessary for allowing higher branch prediction throughput and noncontiguous basic block fetching, limits fetch bandwidth.

The above issues that are emerging as limiting factors in instruction fetching have prompted several significant research studies. A fill mechanism to accomplish noncontigous basic block fetching and alignment has been proposed [6], [7]. In addition, there has been significant work in the area of multiple branch prediction in combination with multiple basic block fetching [2]. These research studies all address the issues of branch throughput and noncontiguous instruction fetching but at the cost of the last factor, namely fetch unit latency. This last issue is addressed with the groundbreaking work in the development of trace caches. A trace cache provides a low latency mechanism for concatenating basic blocks to form instruction traces that are dynamically created during instruction fetching [1].

The trace cache captures and stores instruction sequences, called traces, from the dynamic fetch stream. Later on during program execution, if the same instruction sequence is present in the dynamic instruction stream, then the instruction sequence can be fetched from trace cache. Traces

are specified by a start address as well as the branch outcomes of the control transfer instructions present within it. For a given fetch iteration, if the starting address of a particular trace matches the fetch address, and the predicted outcomes of the branches match the branch directions specified in the trace, then the trace cache provides the trace to the instruction buffer. The basic idea is that the stored instruction traces can cross basic block boundaries, allowing more than one basic block to be fetched in a given cycle. Figure 1-3 shows the high level functioning of the trace cache scheme:

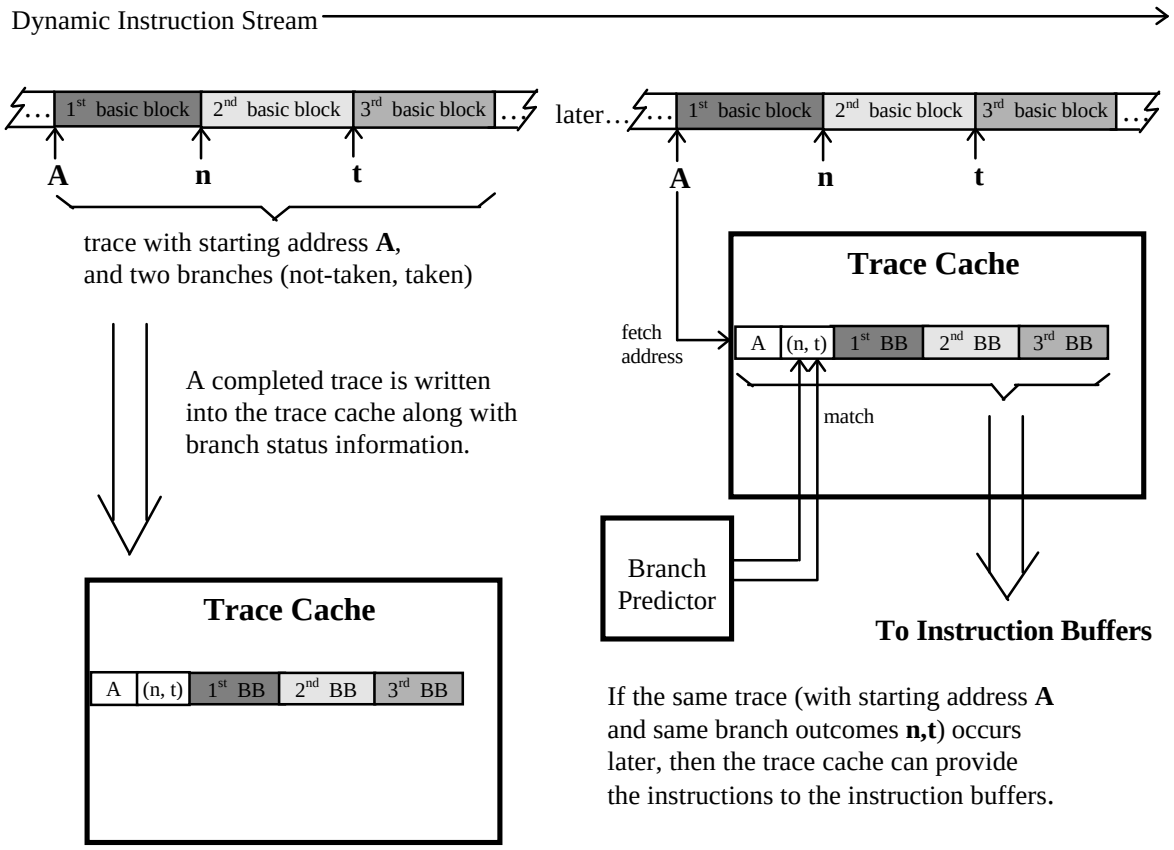

Figure 1-3: Basic Functioning of Trace Cache

The trace cache takes advantage of temporal locality by storing dynamic traces for reuse. The trace cache is accessed in parallel with the instruction cache. Thus, in the worst case of a trace cache miss, normal instruction fetching proceeds through the instruction cache. The basic block concatenation for creating the traces is done off the critical path of the fetch mechanism so latency

is not increased. This is the primary advantage of the trace cache over other noncontiguous basic block alignment mechanisms.

The concept of a trace cache exhibits great potential as a method to increase fetch bandwidth. However, beyond the original design presented in [1], there has been no significant studies that investigate the design space of trace caches. This lack of follow-up research on such a promising idea presents the motivation for this thesis. The research for this thesis attempts to explore the design space of trace caches by examining the performance effects of critical trace cache design parameters. The ultimate goal is to design a high bandwidth instruction fetch mechanism suitable for actual implementation into a future generation microprocessor.

1.2 Thesis Overview

The remaining portion of this thesis is presented in seven chapters. In Chapter 2, relevant previous work is discussed. Chapter 3 describes the basic trace cache design and branch prediction mechanisms. Chapter 4 summarizes the trace cache design issues that will be addressed in the thesis. The simulation methodology is presented in Chapter 5. Simulation results and the ensuing discussions are presented in Chapter 6. Finally, Chapter 7 provides conclusions to this research, followed by a chapter on future work.

Chapter 2

Prior Work

There have been several techniques that has been proposed to overcome the bottleneck presented by control transfer instructions within the dynamic instruction stream. These techniques were developed from a number of recent studies on high instruction fetching bandwidth and is closely related to the research for this thesis. All of these studies deal with proposals for aligning multiple noncontiguous basic blocks from the instruction cache in order to increase instruction fetch bandwidth.

The groundwork for the multiple basic block fetching was done by Yeh, Marr, and Patt, who proposed the idea of a Branch Address Cache (BAC) as a fetch mechanism to increase instruction fetch bandwidth [2]. This mechanism consists of an interleaved instruction cache, a multiple branch predictor, an interchange and alignment network, and the branch address cache itself. The BAC can be viewed as an extension to a branch target buffer, in which the starting address of the next basic block is provided for a single given branch. The BAC is a generalized BTB in the sense that it provides the starting addresses of the next *m* basic blocks, given the multiple branch predictions for the next *m* branches (here, *m* is a small integer set by the implementation). These starting addresses are fed into the instruction cache, which aligns the basic blocks together. A second fetch mechanism proposed by Dutta and Franklin [3] also has a variant of a branch address cache for multiple branch targets. However, they provide an alternative method of single-cycle multiple branch prediction.

A third instruction fetch mechanism was proposed by Conte, Menezes, Mills, and Patel which features a collapsing buffer (CB) [4]. It is composed of an interleaved branch target buffer, an interleaved instruction cache, a multiple branch predictor, and an interchange and alignment network with collapsing buffer logic. Two access iterations are made to the interleaved branch target buffer, allowing two noncontiguous cache lines to be fetched. The collapsing buffer then merges the appropriate instructions from the cache lines together to form the fetch block.

The fetch mechanisms proposed above have a number of disadvantages, namely:

1) An additional pipeline stage before the instruction fetch may be required, because the pointers to all the noncontiguous instruction blocks must be generated before fetch can begin.

2) Additional hardware complexity in the instruction cache because multiple noncontiguous cache lines must be accessed simultaneously.

3) An additional pipeline stage after the instruction fetch may be required, in order to merge and align instruction blocks after they are fetched for the issue buffer.

More recently, another fetch mechanism was proposed by Rotenburg, Bennett, and Smith [1] that eliminates the above disadvantages of the two previously discussed instruction fetching schemes. It involves a special instruction cache called a trace cache to capture dynamic instruction sequences. The trace cache removes the instruction fetch bandwidth bottleneck by storing these dynamic sequences of instructions (i.e. traces) as the program executes via a fill buffer. These traces can be composed of multiple basic blocks and may contain one or more taken branches.

In this scheme, a trace cache would be used in conjunction with a standard instruction cache. The branch predictor of the processor would provide the predicted branch direction of the next few branches in the program, and if they matched the direction of the branches within a stored trace, the trace cache is used to fill the issue buffer instead of the conventional instruction cache. If no match is made, then fetching proceeds in a standard manner from the instruction cache. Instructions from the instruction cache are merged by a fill buffer over time and written to the trace cache when the fill buffer is done with assembling the trace.

The advantage of this setup is the fact that the fill buffer does not lie on the critical path of the instruction fetch, and hence latency through this structure has little impact on overall performance. The basic assumption is that augmenting an instruction cache with a trace cache and associated fill buffer can significantly improve the fetch IPC performance of a microprocessor.

A trace cache only improves performance if dynamic sequences of instructions are commonly re-executed. This seems to be the case, both because of temporal locality, in which instructions used recently have a high probability of being used again, and branch behavior, since most branches are usually heavily biased toward one direction.

There are two other previously proposed hardware mechanisms that exhibit similarities to the trace cache scheme but were developed for other applications. Melvin, Shebanow, and Patt [5] proposed a fill unit mechanism to cache RISC-like instructions from a CISC instruction stream. Franklin and Smotherman [6], [7] extended the functionality of the fill unit to assemble instructions from a fetch stream into VLIW-like instructions to be cached in a structure called a shadow cache. The purpose of the shadow cache is to remove the need for dependency checking in the issue stage, thereby reducing complexity in wide issue processors.

Work on multiple branch prediction is also very relevant, as all of the above schemes rely on it in some form. Recently, Wallace and Bagherzadeh have proposed schemes for multiple branch and block prediction [8]. Since the introduction of the trace cache, path prediction has also become a topic of interest. Path prediction is a form of program execution prediction, where *paths* of execution are predicted. Both Rotenburg [9] and Conte [10] have made contributions to this area of research.

Chapter 3

Trace Cache Fetch Mechanism

This chapter describes the implementation details of the trace cache fetch mechanism discussed in the introduction. It is based on the original trace cache device proposed by Rotenburg, et. al. [1], which forms the foundation for this thesis research. The basic trace cache mechanism is described in detail first, followed by a description of the associated multiple branch predictor.

3.1 Trace Cache

The fetch scheme using a trace cache works in conjunction with a standard instruction cache. At the beginning of each cycle, the trace cache is accessed in parallel with the instruction cache based on the fetch address. The multiple branch predictor generates the branch outcomes for the next couple of branches while the caches are being accessed. If there is a trace cache hit, then the trace is read from the trace cache into the instruction issue buffer. A trace cache hit occurs when the fetch address matches the address tag of a trace *and* the branch predictions match the branch directions of the trace. If either of these conditions is not true, then there is a trace cache miss and normal instruction fetching proceeds from the instruction cache. The complete trace cache fetch mechanism is depicted in Figure 3-1.

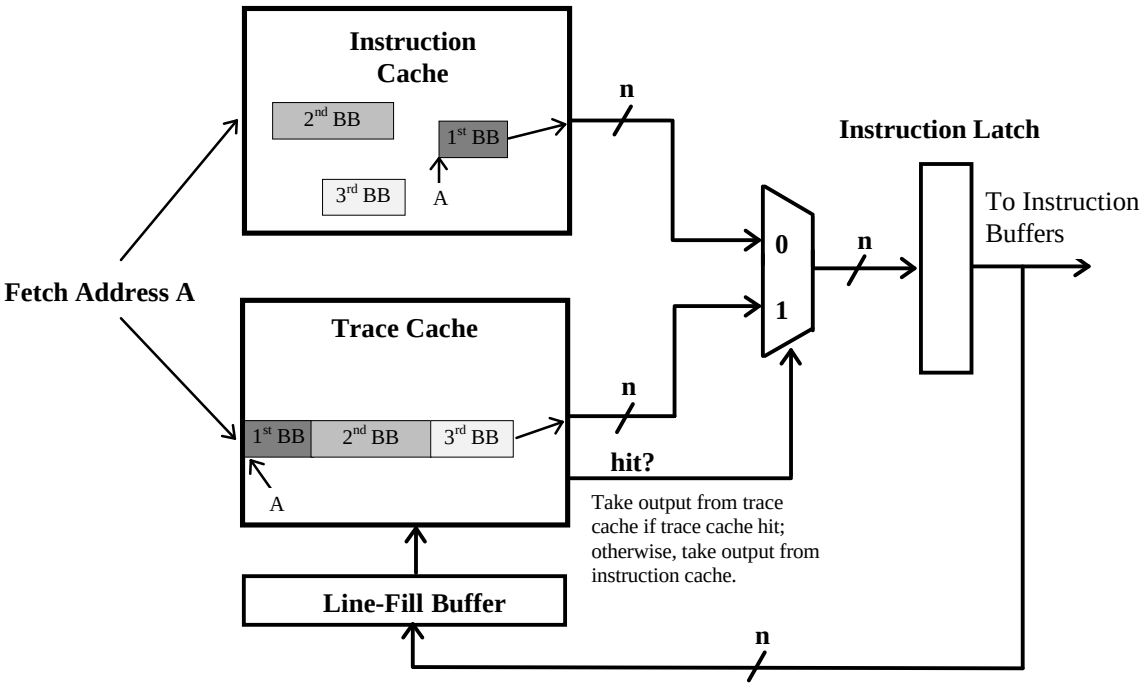

Figure 3-1: Trace Cache Fetch Mechanism

The trace cache itself consists of instruction trace lines, control information, and line-fill buffer logic. The trace lines hold the actual instructions in the trace. The maximum trace length *n* is specified by the trace cache line width. Each trace is also constrained by the maximum number of branches *m* present within the trace. Every trace line has associated with it some control information, namely:

- ***Valid Bit***: Indicates whether or not a trace line contains a valid trace.
- ***Tag***: Identifies the starting address of the trace, i.e. the higher order bits of the address for the first instruction of the trace.
- ***Branch Flags***: A single bit for each branch within the trace to indicate whether or not the branch was taken or not taken. The m^{th} branch of the trace does not need a flag since no instructions follow it. Therefore, only *m* - 1 bits are needed to encode the branch flags.

- ***Branch Mask***: Some state is needed to indicate the number of branches within the trace. This information is used by both the trace cache, to determine how many branch flags to check, and by the branch predictor, for updating. The first $\log_2(m+1)$ bits encode the number of branches present in the trace. The branch mask also has an extra bit to determine whether or not the last instruction of the trace is a branch. If so, the corresponding branch flag does not have to be checked since no instructions follow it.
- ***Trace Target Address***: The next fetch address if the last instruction is a branch and is predicted taken.
- ***Trace Fall-through Address***: The next fetch address if the last instruction to the trace is a branch and is predicted not-taken.

For a 64-entry direct-mapped trace cache with 16 instruction traces (n=16, m=3), 4kbytes is needed for instruction store, and a minimal 712 bytes is used for the control information.

The line-fill buffer is used to latch basic blocks together to form the traces to be written into the trace cache. The basic blocks are latched one at a time into the fill buffer as instruction fetching proceeds. The fill logic merges each new basic block with the instructions already in the line-fill buffer. Filling is completed when either the maximum trace length n or the maximum number of branches in the trace m is reached. When either case occurs, then the contents of the fill buffer are written into the trace cache, along with the control information. The branch flags and mask is generated during the line-fill process, and the trace target and fall-through addresses are computed at the end of the line fill. If the trace does not end in a branch, then the target address is set equal to the fall-through address. The complete trace cache and line fill mechanism is described in the Figure 3-2.

In order to simplify the fill logic for the trace cache, the trace cache simply does not store indirect jumps, returns, or traps. Whenever one of these instructions is encountered, the trace fill truncates at that instruction. In addition, unconditional branches and calls are simply treated as conditional branches that are highly predictable. Thus, no special case is needed to deal with them.

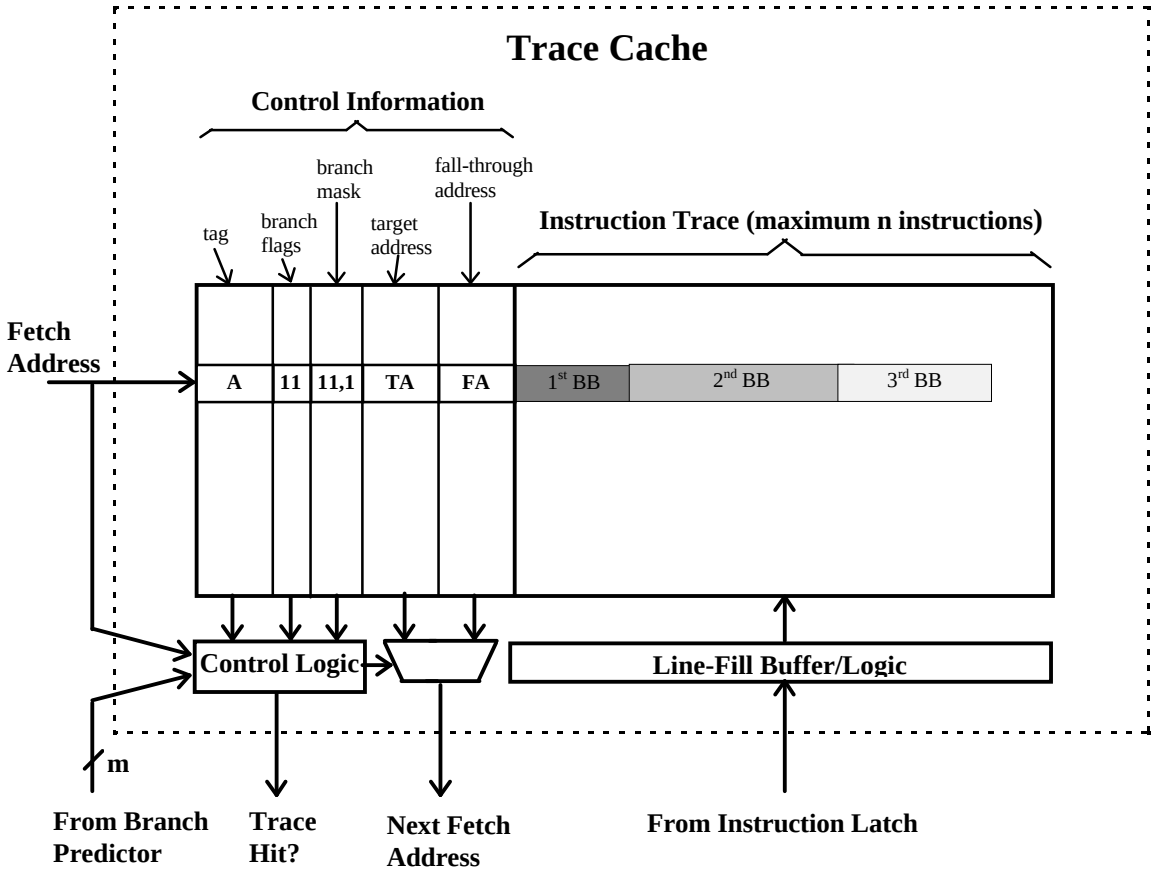

Figure 3-2: Trace Cache Line-fill Buffer Operation with *n*=16 and *m*=3.

In the above figure, the line-fill buffer combines 3 basic blocks fetched from the instruction cache to form the trace. Each basic block is latched into the buffer one at a time. In this case, the branch limit *m* is reached after the third basic block. After this point, the instruction trace and the appropriate control information are written into the trace cache.

The fill logic for the trace cache is complicated by the fact that there are different classes of control transfer instructions that must be handled differently. Control transfer instructions can come as conditional branches, unconditional branches, calls and direct jumps, indirect jumps, returns, and traps. The trace cache design described is only capable of dealing with conditional branches. Indirect jumps, returns, and trap instructions are not included in traces, as these instructions can have an indeterminate number of targets. Since the predictor can only predict one of two targets (the

jump target or fall-through target), the trace cache scheme cannot handle these control transfer instructions. In addition, unconditional branches and calls should not be involved in prediction, since their outcome is known. This complicates the trace cache hit logic, since the trace cache must be able to identify these instructions within the trace and deal with them appropriately.

An issue of importance is the fact that the instruction fill should include the associated delay slot of the branch in question. In Rotenburg's implementation of the trace cache, a basic block ends in a branch and does not include the associated delay slot. Since the delay slot can be assumed as part of the basic block, the implementation that is considered in this research includes the delay slot when possible (i.e. when the maximum trace length has not been reached).

The trace cache design implemented for this thesis research also places the additional restriction whereby a trace cannot start with an instruction that is the delay slot of a branch. If a trace starts with the delay slot of a control transfer instruction, then the control information for the trace cache can inappropriately signal trace cache hits when it is not supposed to. Consider the following situation diagrammed in Figure 3-3. In this scenario, the trace cache will signal a trace hit, since the tag matches the fetch address and the branch directions match the branch predictions. This is the case even if the trace corresponds to *wrong* path of execution, as long as the branch directions of the trace happen to match the predictions. Based on existing control information, there is no way of determining whether the trace represents the correct path of execution. Thus, it is important to prevent traces starting with delay slots from being committed to the trace cache.

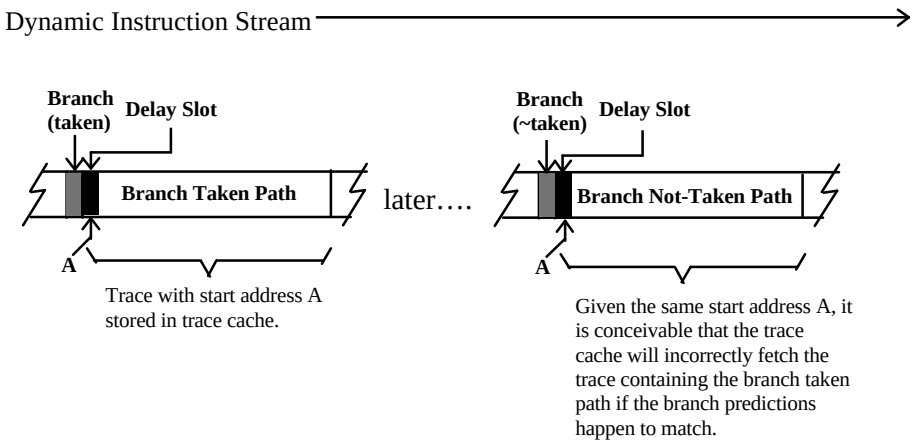

Figure 3-3: Misrepresented Trace Cache Hit Resulting from Starting with Delay Slot

3.2 Multiple Branch Prediction

The trace cache relies on a multiple branch predictor in order to determine whether or not the branch directions of a stored trace match the predicted branch outcomes of the instruction stream. The prediction accuracy of the branch predictor is critical to the performance of the trace cache, as mispredictions result in large trace abort penalties. Thus, the branch predictor used in this research is the Gag correlated branch predictor [21], based on its high prediction accuracy and its ability to be generalized to do multiple branch prediction.

In this scheme, a 16-bit global history register indexes into a pattern history table. The pattern history table actually consists of four sets of counters, which are then selected based on the prediction and history bits. Figure 3-4 shows the basic multiple branch prediction scheme:

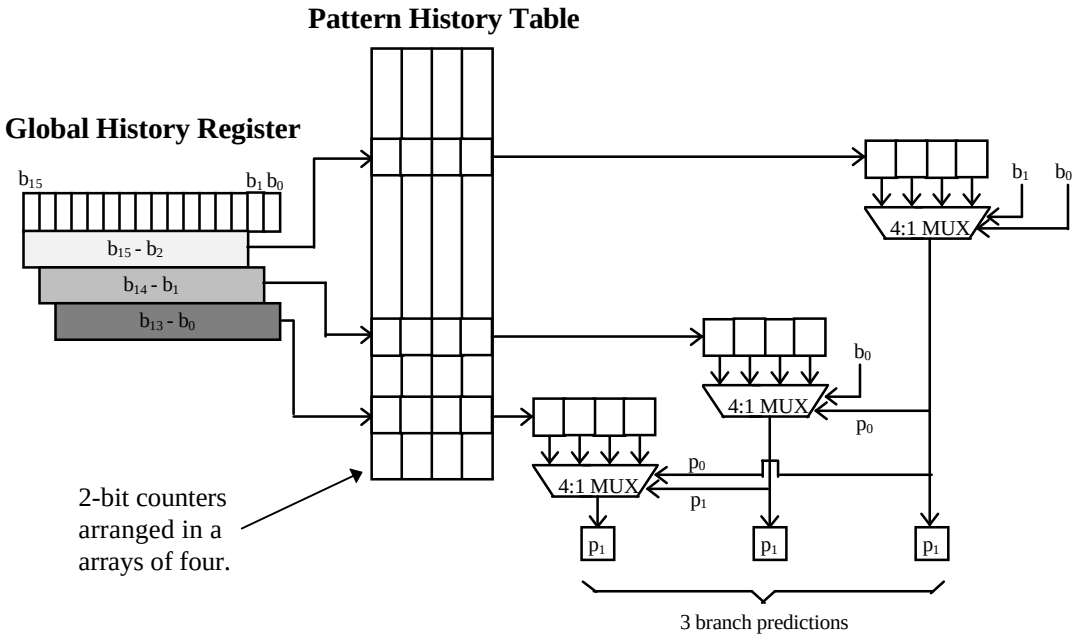

Figure 3-4: Correlated Predictor Scheme Extended To Increase Predictor Throughput

Chapter 4

Contributions of Thesis

The concept of a trace cache shows a great deal of promise, as it provides a method of effectively generating traces for instruction fetching that crosses basic block boundaries without increasing fetch latency. However, this area of research remains relatively untouched and there are a great deal of trace cache design issues that have not been explored. The focus of the research behind this thesis attempts to further improve fetch bandwidth by exploring the design space of trace caches.

This thesis will attempt to refine and develop methods for improving fetch bandwidth beyond limits achievable by currently proposed mechanisms. Specifically, techniques are explored to decrease the trace cache miss rate and increase trace cache storage efficiency. The goal is to establish the viability of the trace cache as a practical hardware device for future high performance microprocessors.

4.1 Trace Cache Design Issues to Address

There is a considerable amount of research that can be done to expand upon Rotenburg, et. al.'s initial work on trace caches. In their paper, the authors only discuss a simple trace cache design. Important issues in trace cache design that this thesis research will address include:

1) ***Indexing***: The simplest way of indexing into a trace cache is by directly using the bits from the fetch address. There are alternative indexing schemes, some of which involve hashing, that may result in more uniform trace distribution within the trace cache.

2) ***Trace Length***: One of the most basic parameters in trace cache design is how long a maximum trace can be. If the trace length is set too high, then trace cache storage efficiency decreases as the extra space within the trace line is wasted. Specifying a trace length that is too

small may also have detrimental effects on trace cache performance by decreasing the effective possible fetch bandwidth. It remains to be seen what the optimal length of a trace line should be.

3) ***Branch Throughput***: The currently accepted standard for multiple branch prediction is limited to a throughput of three. There is no reason to assume that this number is optimal for trace cache fetching. Studies need to be done to see if there is any additional benefit from increased branch throughput.

4) ***Fill Issues***: There are a variety of different methods of how a trace is collected for the trace cache. Specifying how to determine traces for the trace cache can have a significant effect on performance. This includes issues of how the beginning of a trace is specified, how and when instructions are committed to the fill buffer, and how to terminate a trace.

5) ***Partial Matches***: A common situation that arises is when the fetch address matches the start address of the trace but not all of the branch predictions match the path of the trace. In such a situation, it is still feasible for the trace cache to provide instructions to the issue buffer, up to the last valid branch where the prediction matches the taken path. This scheme results in additional complexity, including the additional costs of storing intermediate basic block addresses.

6) ***Associativity***: One of the problems of indexing traces within the trace caches by a start address is the fact that only one trace from a given address can be stored. An alternative method of storage is to increase the trace cache associativity, which could reduce thrashing effects from traces that start with the same address.

7) ***Sizing***: Increasing the size of a trace cache can obviously have a deep impact on performance. However, there has been no research done as to how performance scales with trace cache size.

8) ***Replacement Policy***: As with standard caches, there are several different methods of choosing which trace to replace when filling the cache. It is possible that the replacement algorithm chosen might affect overall trace cache performance.

All the above design issues can be applied to the general trace cache framework proposed by Rotenburg, in which the trace cache acts as a supplementary device to the instruction cache. There also exists the possibility of generalizing the instruction cache by completely replacing it with a trace cache. In this scheme, the trace cache acts as the main mechanism in the instruction fetch,

with a L2 cache used as the source for trace fills. The details of the viability of this scheme presents another focus area for this thesis.

4.2 Long-Term Roadmap of Research

The research for this thesis can be broken down into a number of general steps which are summarized below:

1) Initial statistical gathering and infrastructure development. This includes writing a trace analyzer to collect basic statistics such as average trace lengths, basic block sizes, branch frequencies, number of dynamic basic blocks in a trace, instruction cache hit rates, etc. for standard representative programs.

2) Judiciously selecting metrics suitable in characterizing performance, such as fetch IPC and trace cache miss rate.

3) Implementing a parameterizable trace cache simulator and doing extensive simulating on typical workloads to determine performance improvement of the different trace cache designs over standard instruction fetch mechanisms.

4) Analyzing design issues and iterating refinements based on simulation results. The intent is to refine the original trace cache design as much as possible to maximize trace cache performance.

Chapter 5

Simulation Methodology

The research for this thesis is to be primarily simulation-based and entails the development of a trace-driven simulator in order to gather results and validate trace cache design. In this chapter, the simulation methodology such as the simulation environment, simulation model, performance metrics, and test workload are each discussed in turn.

5.1 Simulation Environment

The simulator development for this research was done under the Mips Technologies Incorporated (MTI) Architecture Simulation Environment. The Architecture Group at MTI is responsible for performance and behavioral simulation for MTI's microprocessor line. In addition, the group is involved with investigation of general architectural innovations for next-generation processors.

In order to facilitate the development of simulators, a well-developed simulation environment was created by the Architecture group. The simulation environment consists of 1) standard benchmark execution trace collections, 2) standard tools for running simulations, and 3) user-provided simulators. The trace cache simulator was developed using this infrastructure. The standard benchmark trace collection is used as the source of simulator inputs, as discussed below under the test workload section. The trace cache simulator is trace-driven, in which an instruction input trace is fed to the simulator for statistics gathering (such as trace cache miss ratio, fetch bandwidth, overall, etc.).

5.1.1 Trace-driven Simulation

The simulation tools run simulations by feeding benchmark execution traces to the simulators. Thus, this type of simulation is called *trace-driven simulation*. Trace-driven simulation is used over other forms or simulation (such as cycle simulation) for a number of reasons. These reasons

include: 1) trace-driven simulation is much faster, 2) representative traces for the SPEC benchmarks already existed, and 3) trace-driven simulation is architecture independent.

The data results collected from this study are all from trace-driven simulation. A distinction must be made between the input traces used in the simulation of the trace cache and the dynamic traces that the trace cache actually stores in the simulator. Trace-driven simulation refers not to the traces stored within the trace cache but to the benchmark application instruction streams that the simulator runs on in order to generate statistics.

There is a problem with trace-driven simulation, namely that incorrect speculation cannot be simulated, as traces represent the actual correct path of execution. This may result in some inaccuracies in the model, as the cache structures do not see the contamination effects of fetching and executing wrongly speculated instructions. However, it is assumed that such effects are negligible and that trace-driven simulation is accurate enough for the purposes of this study.

5.2 Simulation Model

Since this research is focused on increasing instruction fetch bandwidth, the fetch engine of the microprocessor is the primary module of interest and is modeled in detail. The trace cache and instruction cache fetch mechanism, as described in the Trace Cache Mechanism chapter, is implemented under the simulation environment.

In order to isolate the performance effects of the fetch mechanism, it is assumed that the machine stalls for reasons only pertaining to the fetch mechanism and not from issues related to the instruction queue or execution backend. The fetch mechanism can stall on instruction cache misses and incorrect branch predictions.

When using a real branch predictor, an incorrectly predicted branch results in a misprediction penalty. This penalty arises from the need to abort all instructions fetched after the misprediction point. Since the simulator only deals with the fetch portion of the machine (i.e., the execution backend of the processor is not simulated), this mispredict penalty must be approximated. In order to simplify the development of the machine model, all the details of the instruction issue queue and execute engine are abstracted away. For the simulations that involve real branch prediction, the mispredict penalty is simply assumed to be six cycles [2]. This means that instructions following an incorrectly predicted branch will not be fetched until six cycles after the branch is fetched. In

effect, the mispredict penalty embodies all the effects of a mispredicted branch, including aborting the trace and flushing instructions from the instruction queues, etc.

Although the misprediction penalty determination is very simple, it provides an effective way of approximating the effect of branch prediction accuracy on trace cache performance. Using an average misprediction penalty gives a rough estimate on how mispredicted branches affect fetch IPC and is deemed sufficiently accurate for the purposes of this research.

5.3 Performance Metric

The primary performance metric used for this research is fetch IPC (instructions per cycle) from the fetch mechanism. The fetch IPC represents the number of instructions that can be fetched by either the trace cache or primary instruction cache and gives a basic performance gauge of the trace cache and its improvement over using just an instruction cache.

Other metrics of interest include the trace cache miss rate, the miss rate per instruction, and the instruction miss rate. The *trace cache miss rate* indicates the percentage of attempted trace cache accesses that result in a trace cache miss. The *instruction miss rate* represents the trace cache miss rate normalized by the number of instructions fetched. The misses/instruction is usually used as the traditional performance metric for caches. Finally, one can measure the *miss instruction ratio*, or the percentage of instructions fetched not supplied by the trace cache. One goal of this research is to reduce these trace cache miss metrics to the lowest level possible.

5.4 Test Workload

In order to determine the performance of the trace cache fetch mechanism, a suite of programs representative of actual program behavior must be chosen to base performance measurements on. The SPEC benchmark suite is the most widely recognized and used benchmark suite in industry as well as academia, and thus will be used in this research as the primary benchmark source to gauge performance. The SPEC benchmark suite consists of eight integer benchmarks written in C, and ten floating point benchmarks written in Fortran. A brief description of each benchmark is provided in Table 5-1 below:

Benchmark	Description
SPEC95, INT	
099.go	Artificial Intelligence, plays the game of Go.
124.m88ksim	Motorola 88K RISC CPU simulator, runs test program.
126.gcc	New Version of GCC, the GNU C Compiler, builds SPARC code.
129.compress	Compresses and uncompresses file in memory using Lempel-Ziv coding.
130.li	Lisp interpreter.
132.ijpeg	Graphic compression and decompression. (JPEG)
134.perl	Manipulate strings (anagrams) and prime numbers in Perl.
147.vortex	A database program.
SPEC95, FP	
101.tomcatv	Tomcatv is a mesh generation program.
102.swim	Shallow Water Model with 513 x 513 grid.
103.su2cor	Quantum physics. Monte Carlo simulation.
104.hydro2d	Astrophysics. Hydrodynamical Navier Stokes equations
107.mgrid	Multi-grid solver in 3D potential field.
110.applu	Parabolic/elliptic partial differential equations.
125.turb3d	Simulate isotropic, homogeneous turbulence in a cube.
141.apsi	Solve weather conditions (temp., wind, velocity) and distribution of pollutants.
145.fpppp	Quantum chemistry
#146.wave5	Plasma physics. Electromagnetic particle simulation.

Table 5-1: Description of SPEC95 Benchmarks

5.4.1 General Characteristics of SPEC95 Benchmarks

The integer programs represent the bottleneck in instruction fetching, resulting from an average basic block size of around 5. The dynamic basic blocks of the floating-point benchmarks are all significantly larger than the corresponding numbers for the integer benchmarks. This is because the floating-point benchmarks contain more scientific-based code and thus have large parallel loop structures to iterate computations. This is in contrast with the tighter, more complex looping behavior of integer benchmarks. Among the integer benchmarks themselves, the most interesting ones to look at are 099.go and 126.gcc because they exhibit the most erratic behavior and stress the ability of the fetch mechanism to fetch instructions.

5.4.2 Trace Generation

The benchmarks in the SPEC95 suite range from between 25 billion to 90 billion instructions, making it virtually impossible to run the simulator through the complete inputs given the computation resources. Thus, it is necessary to generate a set of reasonable length instruction traces that

are still representative of the original benchmarks. There are two strategies for generating traces of reasonable size. The first involves using input sets that result in shorter benchmark run times. The second is to sample through the actual full trace at even intervals (such as recording 10 million instructions, then skipping the next 90 million instructions and so on). The problem with the first approach is that reducing the problem size may significantly alter the program's behavior on the system. The problem with the second method is that making tradeoffs between accuracy and trace size is non-trivial.

Because it is paramount that the input traces encapsulate actual program behavior, the traces used for this research were created by sampling. The sampling strategy was tweaked for each application benchmark within the SPEC95 suite in order to capture the significant behavioral aspects of each program.

Profiling information for each benchmark was obtained by utilizing the R10000 hardware performance counters when running the benchmarks on a R10000 processor system. This profiling includes basic block, instruction, and instruction class distributions as well as data and instruction cache misses, branch mispredicts, and TLB misses.

Some benchmarks (specifically 099.go and 126.gcc) show dramatic changes in behavior throughout their program runs. For these benchmarks, an ordinary on/off sampling strategy is used. Special attention was given to the periods in the program runs that exhibited the most variation by taking more samples in those periods. For example, 126.gcc does not show any regular behavior for any of its inputs, so a number of the larger inputs were chosen and evenly sampled. In contrast, 099.go shows more drastic changes in the earlier parts of the program. Therefore, the benchmark was divided into 4 sections, with the earlier sections being smaller, and a specified number of 10-million instruction samples were evenly obtained from each section.

Other benchmarks have a cyclic pattern, with their periods comparable to trace lengths that a simulator could execute in reasonable amount of time (around 200 million instructions). For these benchmarks, one or more slices of 200 million instructions were taken consecutively so that one entire cycle of behavior is captured. The programs that exhibit cyclic behavior include the integer benchmarks 129.compress and 132.ijpeg and the FP benchmarks 147.vortex, 102.swim, 104.hydro2d, 107.mgrid, 110.applu, 125.turbo3d, 146.wave5. For a couple of these benchmarks (namely 147.vortex and 146.wave5), the periods of their cyclic behavior was larger than 200M

instructions. For 147.vortex, sampling was done in chunks around the interesting areas whereas for 146.wave5, the behavior was captured over two trace slices.

Finally, some benchmarks do not exhibit any sort of pattern but have different periods of varying behavior. In this case, each period is sampled by different strategies such as in variable length chunks, or on/off sampling, etc. Benchmarks that exhibit such behavior include 124.m88ksim, 130.li, 134.perl, 101.tomcatv,103.su2cor, 141.apsi, and 145.fpppp.

By applying a sampling strategy in this way, only about one billion instructions were needed to generate the sampled traces which represent the entire 717 billion instruction run of the SPEC95 reference set. These traces were created by the MTI Architecture Group and are used as the source of instruction traces for this research. Since the trace samples represent less than 0.15% of the actual benchmark runs, it is important to verify that the sampled traces actually represent actual benchmark behavior. To provide an independent check on how representative the sampled traces are, the profile information for a complete program run was compared to the behavior of the sampled traces. Since the profile results for the sampled trace matched the distributions for the actual program run, there is some assurance that the samples are accurate representations of actual code execution. Table 5-2 lists the benchmarks traces that are used in this study.

Integer Benchmarks	Number of Traces (~200M Inst. each)	Floating Point Benchmarks	Number of Traces (~200M Inst. each)
099.go	2	101.tomcatv	3
124.m88ksim	2	102.swim	1
126.gcc	2	103.su2cor	3
129.compress	2	104.hydro2d	2
130.li	3	107.mgrid	1
132.ijpeg	3	110.applu	2
134.perl	2	125.turb3d	3
147.vortex	4	141.apsi	1
		145.fpppp	3
		146.wave5	3

Table 5-2: Traces for Benchmarks

Chapter 6

Simulation Results and Analysis

In this chapter, the general results of the thesis research are presented. Using the simulation methodology described in the previous chapter, the effects of various parameters on trace cache performance and efficiency are explored. The results section starts by presenting the simulation results for a conventional instruction fetching mechanism, as compared with a base trace cache design. Different aspects of the trace cache design are then explored in turn. The section concludes with the effects of using a real branch predictor.

6.1 Instruction Cache Results

As a base method of comparison, the trace cache performance results are compared to a conventional instruction cache fetch mechanism. Table 6-1 summarizes the instruction cache parameters used as the comparison base:

Primary Instruction Cache	
Size	128KB
Block Size	64B
Associativity	2
Replacement	LRU
Miss Penalty	10 cycles

Table 6-1: Instruction Cache Parameters

For these simulations, the maximum fetch bandwidth is set at 16 instructions/cycle. The resulting fetch IPC values when just using an instruction cache are show in Figure 6-1.

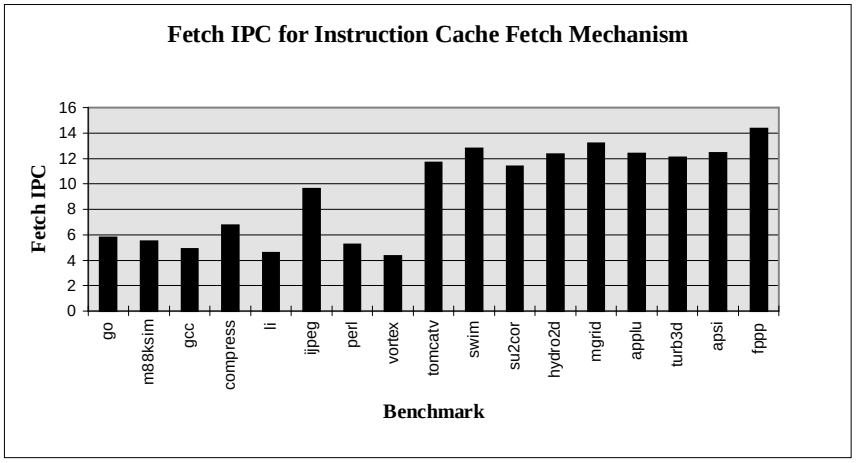

Figure 6-1: Fetch IPC for Instruction Cache Fetch Mechanism

From the figure, we notice several things immediately. For the integer benchmarks (first 8 benchmarks), the average fetch IPC out of the instruction cache is comparable to the average basic block size of the benchmark that was run. This is indicative of the fact that the basic block size is the limiting factor in the fetch IPC, as the instruction cache can only fetch up to one basic block at a time. This would mean for most common non-scientific applications, the bandwidth ceiling is limited to an average of 5-6 instructions per cycle, regardless of the parallelism of functional units that might exist.

The fetch IPC values for the floating-point benchmarks are much higher than the corresponding values for the integer benchmarks. Since the branch frequencies are much smaller for these benchmarks than the integer benchmarks, the instruction cache can fetch more instructions in a given cycle. This results in fetch IPC values that are much closer to the maximum fetch limit.

In the next section, we will see that by simply adding a trace cache in conjunction with the instruction cache, we can significantly improve the fetch IPC and hence overall performance of the microprocessor.

6.2 Base Trace Cache

The results derived above for the instruction cache fetch mechanism are first compared to that of the most basic trace cache design. For initial measurements, the original trace cache model from Rotenburg's design is used, as described in his paper [1]. This basic trace cache design has the following parameters:

Base Trace Cache Model	
Trace Entries	64
Trace Length	16 Instructions
Max. # Brs.	3
Associativity	1 (direct-mapped)
Overall Size	4 Kbytes

Table 6-2: Base Trace Cache Design

For these simulations, the maximum fetch rate for the simulator is equal to the trace cache line size, which is equal to 16 instructions in this case. Also, in order to isolate the effects of the trace cache itself, perfect branch prediction is assumed for the present. The results of the simulation runs are shown in Figure 6-2:

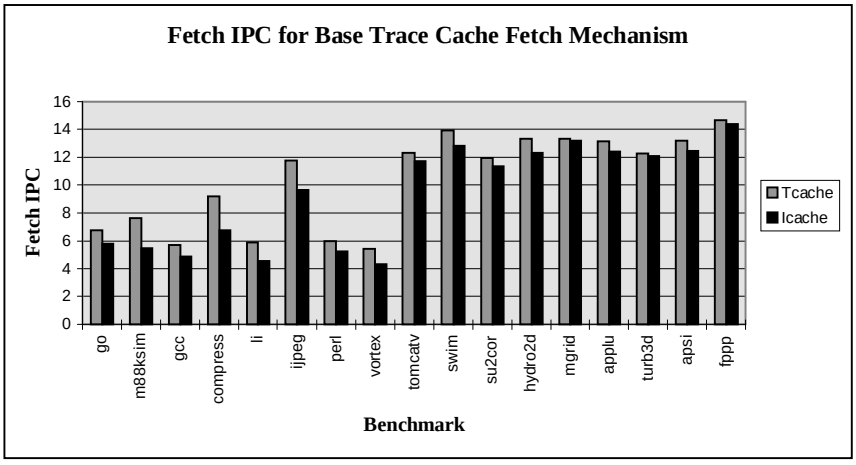

Figure 6-2: Fetch IPC for Base Trace Cache Fetch Mechanism

From the fetch IPC results for the basic trace cache design, we see immediately that augmenting the instruction cache with a small trace cache results in considerable fetch performance improvements. Figure 6-3 summaries the performance improvement of the trace cache fetch mechanism over a traditional instruction cache fetch device:

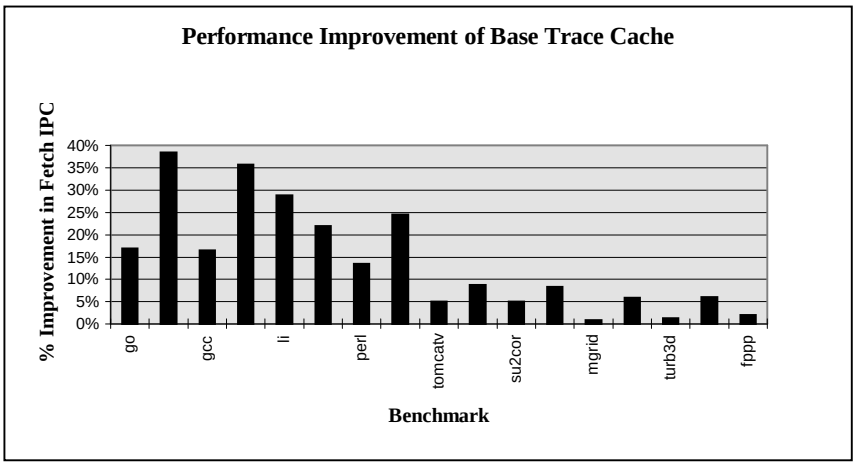

Figure 6-3: Performance Improvement of Base Trace Cache

From Figure 6-3, it is apparent that the trace cache improves the performance of the integer benchmarks much more than the floating-point benchmarks. In fact, a number of the floating-point benchmarks do not improve much at all. The reason is the trace cache only removes the limitation of fetching a single basic block. For the integer benchmarks, a basic block is only 5-6 instructions in length on average, so fetch IPC improves dramatically. However, the average basic block size for the floating-point benchmarks is already larger than the maximum fetch limit (16 instructions), so only marginal improvements in the fetch IPC can be made. The statistics for the average trace that is written into the trace cache and the average trace that is actually read from the trace cache are summarized in Table 6-3.

The statistics give an gauge as to how the instruction traces are being utilized by the fetch mechanism. From the table, it seems that there is not too much variation between the traces that are written into the trace cache and what is actually used by the fetch mechanism. The lack of variation is a good indication that the traces stored in the trace cache are being equitably utilized by the fetch mechanism. Were this not the case, such as if the trace cache only used short traces or traces containing only one or two basic blocks, then there would be a greater disparity between the written and read trace statistics.

INTEGER	go	m88ksim	gcc	compress	li	ijpeg	perl	vortex	
trace written	11.3	11.0	10.2	10.7	10.8	13.1	11.6	9.3	
basic blks written	1.94	1.73	1.92	1.66	2.05	0.82	1.82	2.09	
trace read	10.4	9.5	9.9	11.3	9.4	11.7	11.0	9.0	
basic blks read	1.83	1.86	1.77	1.53	1.91	0.97	1.90	2.02	
FP	tomcatv	su2cor	swim	hydro2d	mgrid	applu	turbo3d	apsi	fppp
trace written	13.8	14.3	13.0	15.1	15.0	14.0	12.0	13.6	15.3
basic blks written	0.58	0.34	0.87	0.22	0.15	0.59	0.83	0.64	0.18
trace read	12.3	13.6	13.6	12.8	13.2	13.2	12.9	13.4	13.5
basic blks read	0.80	0.32	0.88	0.53	0.34	0.69	0.63	0.62	0.55

Table 6-3: Average Written/Read Trace Statistics

It is pleasantly surprising to note that the average trace lengths for the integer benchmarks are actually on par with that of the floating-point benchmarks. This fact shows how effective a trace cache is in improving the fetch bandwidth of the integer benchmarks. However, the traces for the integer benchmarks contain, between 1.5 to 2 basic blocks on average. This is in contrast to the floating-point benchmarks, which only store on average around half of a basic block. This conforms to expectations, as the floating-point benchmarks have much larger basic blocks. Sections 6.3-6.7 describe results for attempts to further improve these trace statistics.

The final results of interest for the base trace cache design are the miss statistics. These trace cache miss rates are shown in Figure 6-4. The miss rates for the basic trace cache design are abysmal. For a majority of the benchmarks, more accesses occur from the instruction cache than from the trace cache. This is a noted problem with the implementation proposed by Rotenburg in his original paper. The corresponding instruction miss rates are an order of magnitude off from miss rates for current instruction caches.

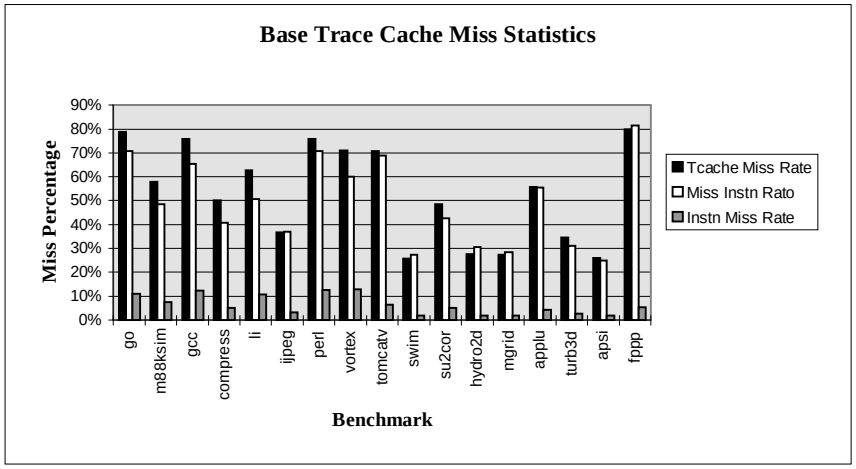

Figure 6-4: Base Trace Cache Miss Statistics

One possible fetch mechanism design optimization is to have the trace cache completely replace the instruction cache as the primary fetching device, with a L2 cache used as a fill mechanism for the trace cache. However, it is not possible to realize such an implementation using the original trace cache design. The instruction miss rates are much too high, which would result in unacceptable performance losses given the miss penalty. Figure 6-5 shows the breakdown of the cause of a trace cache miss:

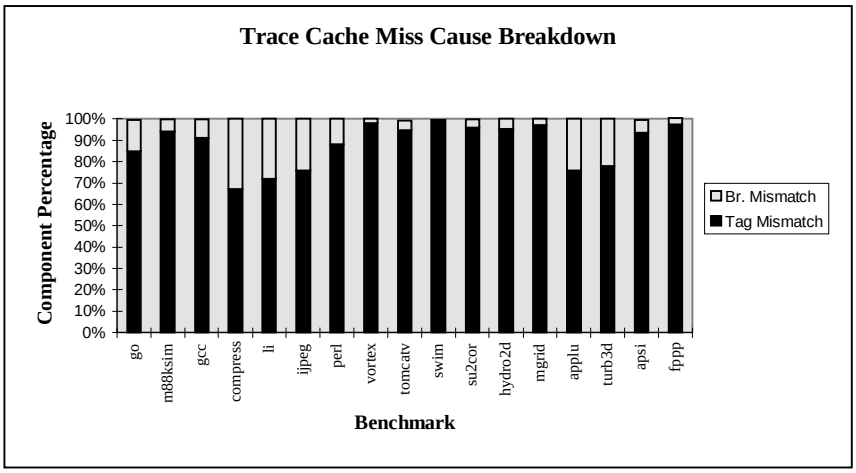

Figure 6-5: Trace Cache Miss Cause Breakdown

A trace cache miss occurs if either 1) the trace cache tag doesn't match the fetch address or 2) the branch flags do not match the branch predictions. From the breakdowns, we see that most of the trace misses result from tag mismatches, indicating either that there are a great deal of collisions within the trace cache, or that the trace cache is simply too small to store enough useful traces. Both these problems can be rectified by increasing the associativity and/or size of the trace cache, as will be shown in sections 6.8 and 6.9 respectively.

6.3 Effects of Hashed Indexing

Given the base trace cache results, this research first explores the possibility of altering the indexing scheme of the trace cache to improve performance. The multiple branch predictor that is necessary as part of the trace cache fetch mechanism represents a very large amount of space overhead. As an alternative to having a branch predictor, one can have the trace cache indexed as a hash of the global history register and the fetch address (such as an XOR function). This well-known scheme is known as the gshare indexing method [18], used originally to index into a branch prediction table. The gshare indexing methodology can be adapted for use with the trace cache.

Using this indexing scheme, a trace cache hit is then simply based on whether or not the fetch address matches the tag of the trace. There are several aspects of this design that significantly simplifies the trace cache fetch mechanism. First of all, trace status information is reduced only to a tag, reducing trace storage overhead. In effect, the branch global history information that is used to derive the index is used as the "branch prediction" of the trace. Thus, the branch prediction hardware can be completely removed from the fetch mechanism, saving a great deal of space and complexity. Also, because of the nature of the XOR hash performed to derive the index, traces can be more uniformly spaced within the trace cache. Figure 6-6 depicts the hashed indexing into the trace cache.

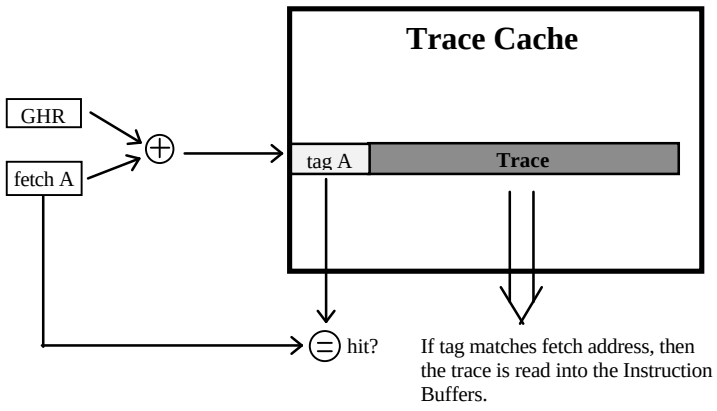

Figure 6-6: Hashed Indexing into the Trace Cache

The resulting fetch IPC for this indexing scheme is shown in Figure 6-7:

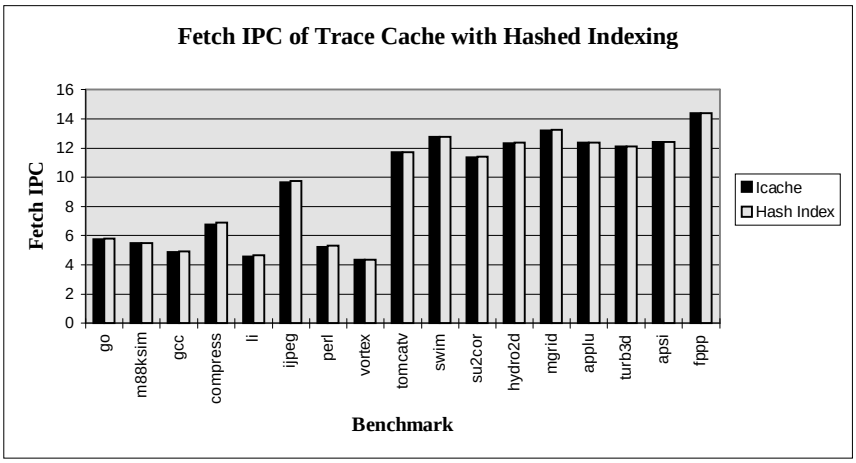

Figure 6-7: Fetch IPC of Trace Cache with Hashed Indexing

From the fetch IPC results, we see that hashed indexing into the trace cache does not work very effectively at all. The resulting IPC values are far worse than using the original indexing scheme, and marginally greater than just using an instruction cache. The main reason is the extremely bad trace cache miss rates using the XOR hash, as shown in Figure 6-8:

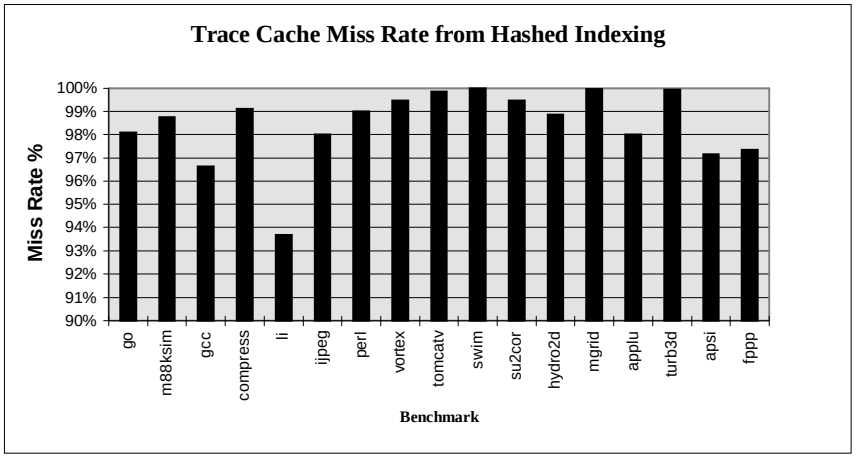

Figure 6-8: Trace Cache Miss Rate from Hashed Indexing

Although hashed indexing is not a bad idea in and of itself, we see that unless we improve the resulting trace cache miss rate, the idea is not very feasible.

6.4 Effects of Trace Line Length

The trace line length is one of the basic parameters of the trace cache, dictating the maximum length of the trace that can be stored. Since longer traces can be stored, it is intuitive that increasing the line size would increase trace cache performance. However, it is necessary to explore how performance scales with increased trace length and whether or not this increase in performance is commensurate with the additional storage cost of increasing the maximum trace length. Since the maximum trace length specifies exactly how much storage is set aside for a single trace regardless of how long the actual trace is, setting an overly long trace line size would result in wasted storage.

The maximum line constraint can also feasibly affect performance based on the way existing traces are incorporated into a trace fill. Consider the case where a trace hit occurs while the fill buffer is still collecting instructions for a trace fill. Only complete traces can be stored in the fill buffer during the fill process; no partial traces fills are allowed since it is not possible to determine the number of control instructions in the partial trace given existing control information. This, coupled by the fact that a trace cache hit occurs relatively frequently, results in the possibility of having only small traces stored. This can occur when a fill starts with a basic block, followed by a long trace. If both the basic block and trace cannot fit in the fill buffer, only a single basic block will be stored. By increasing the maximum trace length, this possibility is reduced.

Increasing the trace length size beyond 16 has a modest effect on the fetch IPC performance of both the integer and floating-point benchmarks. However, the performance curve for the integer benchmarks tapers off with larger trace line lengths whereas the floating point benchmarks still improve slightly. Figure 6-9 shows the integer fetch IPC results when the maximum trace length is varied from between 10 to 24 instructions, in increments of 2.

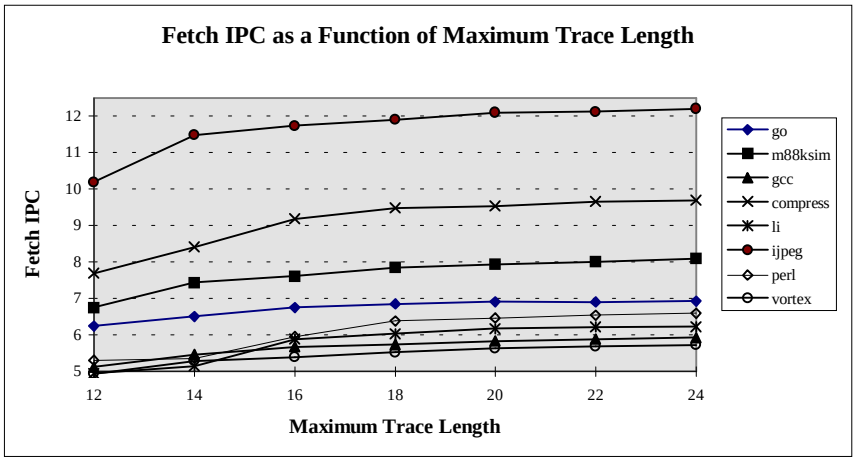

Figure 6-9: Fetch IPC as a Function of Maximum Trace Length

From Figure 6-9, we see that integer fetch IPC monotonically increases for trace lengths under 20, after which point improvement levels off. Given an average basic block size of 5 instructions, a branch throughput limit of 3 would result in a trace just about the size of the maximum trace length for the base trace cache design. The fact that fetch IPC improves for longer trace lengths seems to indicate that the variance of basic block sizes could result in situations where increased trace line size is beneficial. It is possible that further IPC improvement with trace line lengths greater than 20 may be caused by a limitation of the branch throughput. However, in the next section, we will see that this is not the case. The fetch IPC improvement for a trace line length of 20 is shown in Figure 6-10:

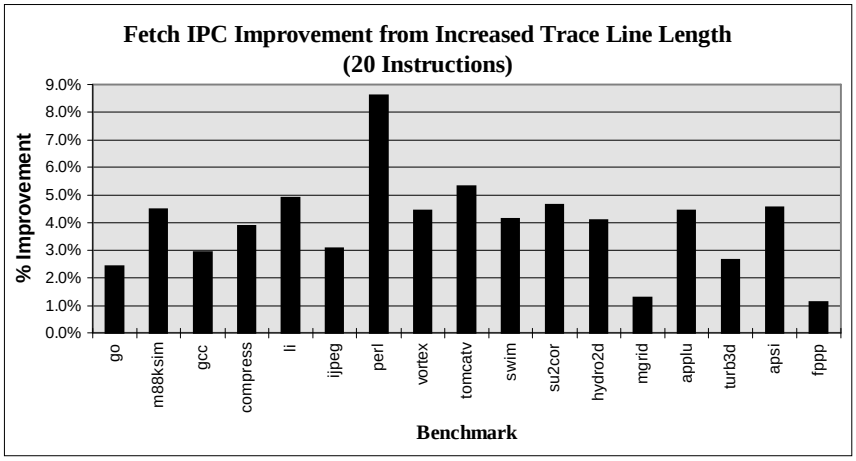

Figure 6-10: IPC Improvement from Increased Trace Line Length

The corresponding average trace lengths that are written/read are shown in Table 6-4:

INTEGER	go	m88ksim	gcc	compress	li	ijpeg	perl	vortex	
trace written	12.1	11.6	10.9	11.2	11.4	13.3	12.8	9.9	
basic blks written	2.05	1.82	2.02	1.71	2.21	0.96	1.99	2.14	
trace read	11.7	10.1	10.4	11.8	9.8	12.1	12.1	9.6	
basic blks read	1.96	2.00	1.83	1.66	2.13	1.04	2.07	2.10	
FP	**tomcatv**	**su2cor**	**swim**	**hydro2d**	**mgrid**	**applu**	**turbo3d**	**apsi**	**fppp**
trace written	14.1	14.7	13.3	15.9	15.1	14.4	12.6	14.2	15.5
basic blks written	0.64	0.37	0.95	0.28	0.22	0.67	0.86	0.69	0.23
trace read	13.1	14.1	13.9	13.4	13.3	13.8	13.5	13.7	13.6
basic blks read	0.85	0.35	0.93	0.57	0.46	0.72	0.70	0.67	0.61

Table 6-4: Average Trace Written/Read Statistics for Trace Line Size = 20

We see that increasing the maximum trace line size results in an increase in both the trace length written into the trace cache as well as the average trace length read from the trace cache during a hit.

6.5 Effects of Increasing Branch Throughput

Based on the results from the preceding section, it is legitimate to ask whether or not increasing branch throughput, and hence the number of basic blocks that can be stored in the trace cache, would further improve fetch IPC.

In order to support additional branch throughput, several modifications are necessary to both the trace cache and branch predictor. The trace cache modifications amount to simply adding branch flags and branch number bits to the trace cache control information. These changes are very straightforward and do not add much to the implementation cost. However, the changes to the multiple branch predictor represent a much larger overhead. Increased branch throughput support entails adding an additional set of counters to the multiple branch predictor. Thus, the pure hardware cost is very large for each additional branch prediction that is to be added. In addition, there is no reason to assume that the proposed multiple branch prediction strategy would scale well beyond three predictions. Thus, increasing branch throughput can possibly affect branch prediction accuracy as well as increasing the space overhead.

Because of the large space overhead, there must be significant performance gains in order to justify increasing branch throughput. However, simulation results indicate that increasing branch throughput does not have any significant effects on fetch IPC. The effects of varying branch throughput between 1 to 5 for the integer benchmarks are shown in Figure 6-11. For these simulations, the maximum trace line length is set to infinity in order to prevent the line length from being the limiting factor.

From Figure 6-11, we see that increasing branch throughput is highly effective for improving integer fetch IPC only past the third branch. Any additional branch throughput does not improve fetch IPC at all. Simulations show the same trend for the floating-point benchmarks. These results are not surprising; there is a diminishing return from increased throughput.

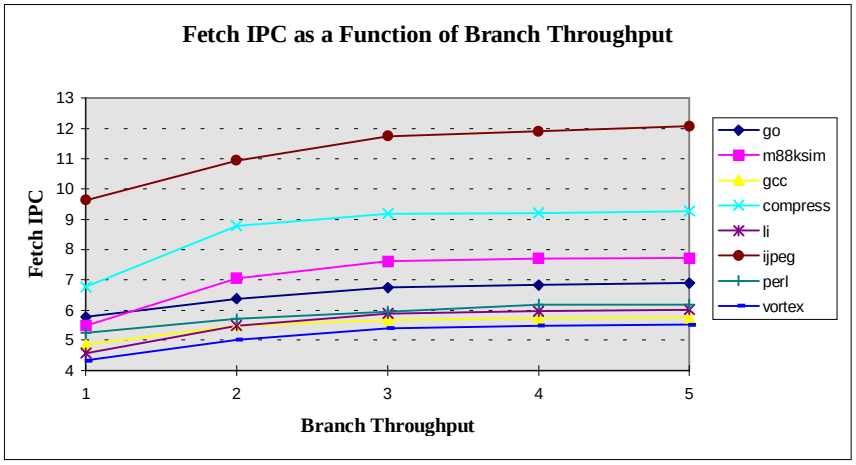

Figure 6-11: Integer Fetch IPC as a Function of Branch Throughput

Storing traces that contain more than 3 basic blocks are not utilized as often, as they represent more specific paths of execution. This fact can be verified by looking at the written/read trace statistics for the higher branch throughput simulations, as shown in Figure 6-12:

INTEGER	go	m88ksim	gcc	compress	li	ijpeg	perl	vortex	
trace written	12.6	12.1	11.8	12.2	11.6	13.9	12..9	10.4	
basic blks written	2.10	1.96	2.25	2.01	2.69	1.32	2.30	2.41	
trace read	10.9	9.7	9.9	11.4	9.7	12.0	11.3	9.4	
basic blks read	1.86	1.89	1.75	1.58	1.95	1.03	2.07	2.22	
FP	**tomcatv**	**su2cor**	**swim**	**hydro2d**	**mgrid**	**applu**	**turbo3d**	**apsi**	**fppp**
trace written	13.9	14.1	13.3	15.4	15.1	14.4	12.2	13.9	15.4
basic blks written	0.61	0.32	0.92	0.24	0.17	0.60	0.86	0.65	0.18
trace read	12.7	13.6	13.9	13.2	13.4	13.8	13.4	13.7	13.7
basic blks read	0.82	0.37	0.91	0.57	0.33	0.72	0.69	0.66	0.56

Figure 6-12: Average Written/Read Trace Statistics for Branch Throughput = 5

Whereas the statistics for the floating-point benchmark are much the same as before, the corresponding figures for the integer benchmarks are slightly different. The trace written statistics show that for the integer benchmarks, the trace cache stores longer traces consisting of more basic blocks. However, the trace read statistics illuminate why fetch IPC does not actually improve.

The trace read statistics are almost identical to that of the base trace cache design, indicating that the trace cache preferentially fetches the shorter traces. Again, this may be because of the fact that the traces that contain more basic blocks cannot be fetched unless all the branch predictions match. Since these conditions are more restrictive, they are not fetched as often. Thus, according to the simulation results, a branch throughput of 3 is perfectly adequate for fetching purposes. One way of relaxing the trace hit requirements is to allow partial hits, as will be shown in section 6.7.

6.6 Effects of Modifying the Fill Mechanism

Looking at the trace cache miss rates for the base design, we can see that the trace cache is not very efficient at all. It is obvious that we can further improve the performance of the trace cache by reducing the miss rates. There are several issues related to the fill mechanism that can possibly be changed to improve trace cache performance.

6.6.1 Storing Only Complete Basic Blocks

The original trace cache design allows trace fills up to the third branch that it encounters or up to a maximum of 16 instructions, whichever comes first. There is an argument that can be made to only include instructions that end on a basic block boundary. First of all, this results in the next fetch IPC to be the start of a basic block. This restriction works well as it allows the trace cache to be accessed; if a delay slot is the start address, then the trace cache must miss as a trace cannot start with a delay slot (see Chapter 3).

However, the most compelling reason to limit trace fills to include only complete basic blocks is to reduce the amount of storage needed to store trace information. Given a loop, it is conceivable that multiple traces are generated from offset addresses. A reason may be that the trace length(s) results in an access stride that does not match the loop length. This mismatch can cause a great deal of unnecessary trace storage for a single loop, as well as cause collisions in the trace cache. This effect is depicted in Figure 6-13.

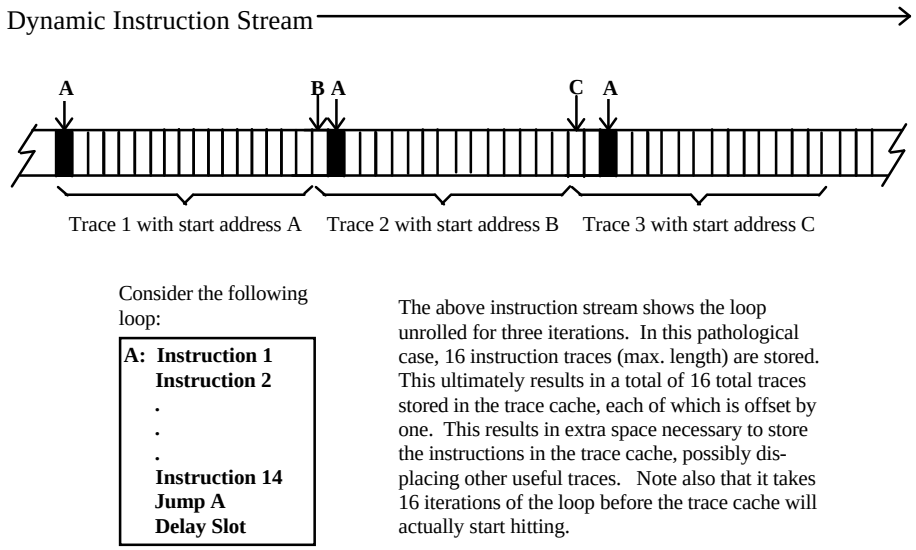

Figure 6-13: Unnecessary Trace Storage from Offset Traces

The above example shows a pathological case where every stored trace is just one instruction offset from each other. This means 16 different traces are stored in the trace cache for the same loop, resulting in highly redundant storage. In this scenario, the trace cache will also be forced to initially miss the first 16 iterations of the loop while all the traces are being generated.

By a simple modification to the trace fill logic for the trace cache, instructions can be committed to the fill buffer only when the end of a basic block is encountered (i.e., the delay slot of a control transfer instruction). During the beginning of a trace fill, the fill buffer initially commits all instructions that are fetched. After the first complete basic block is committed, all following instructions must wait until a complete basic block is available before committing to the fill buffer. Although this complicates the fill logic, the performance improvement from treating basic blocks as atomic units is considerable. Figure 6-14 summarizes the fetch results for a 64-entry trace cache identical to the base trace cache design with this modification.

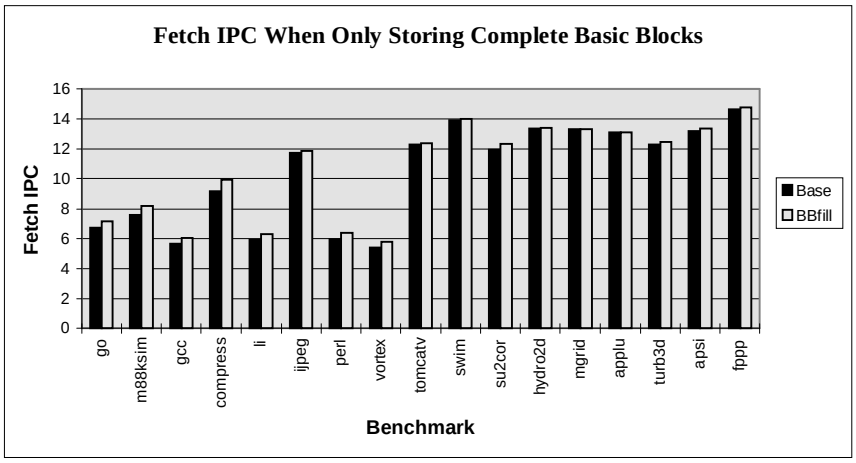

Figure 6-14: Fetch IPC When Only Storing Complete Basic Blocks

From Figure 6-14, we see that the modification to the fill mechanism has the greatest affect on the integer benchmarks. For most of the integer benchmarks, limiting the trace fill to complete basic blocks results in considerable performance gains. In contrast, the floating-point benchmarks results are not affected much at all. The performance improvement is shown in Figure 6-15:

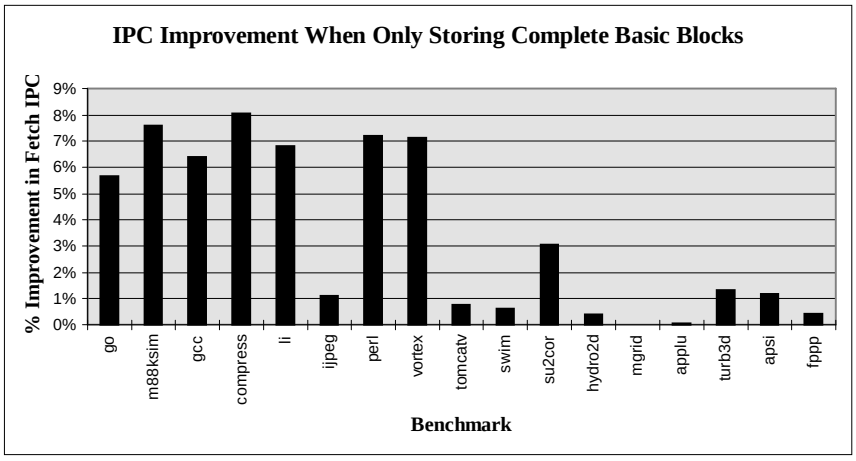

Figure 6-15: Improvement When Only Storing Complete Basic Blocks

The corresponding miss rate statistics are as follows:

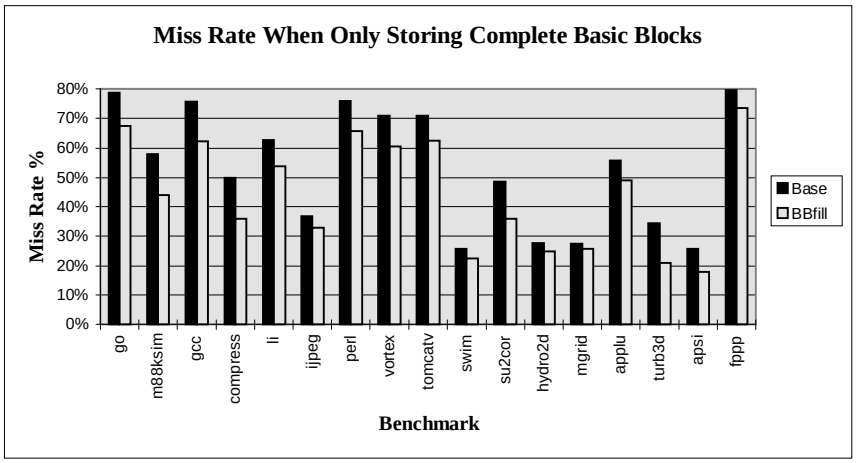

Figure 6-16: Miss Rate When Only Storing Complete Basic Blocks

We see that the trace cache miss rates also drop considerably using the modified fill mechanism. Based on the reasoning explained above, it seems valid that the reduced storage overhead and associated effects of only storing complete basic blocks would result in the lowered miss rates. The decrease in the miss rates, between 13% to 46% for the integer benchmarks, is most likely the predominant reason for the fetch IPC improvement.

6.6.2 Aborting a Trace Fill on a Trace Hit

Another possible fill mechanism modification is to not include traces as part of a trace fill. With this proposed modification, a trace fill terminates when a trace hit occurs. In the original trace cache design, it is possible for a trace fill to include a previously existing trace if a trace cache hit occurs during the fill. This reduces the efficiency of the trace cache as now two identical traces are present in different locations within the trace cache, except one is prefixed by some instructions that were committed to the fill buffer prior to the concatenation of the trace. This may not be a bad thing, depending on how long the prefix length is, but redundancy is still present. Figure 6-17 illustrates this effect:

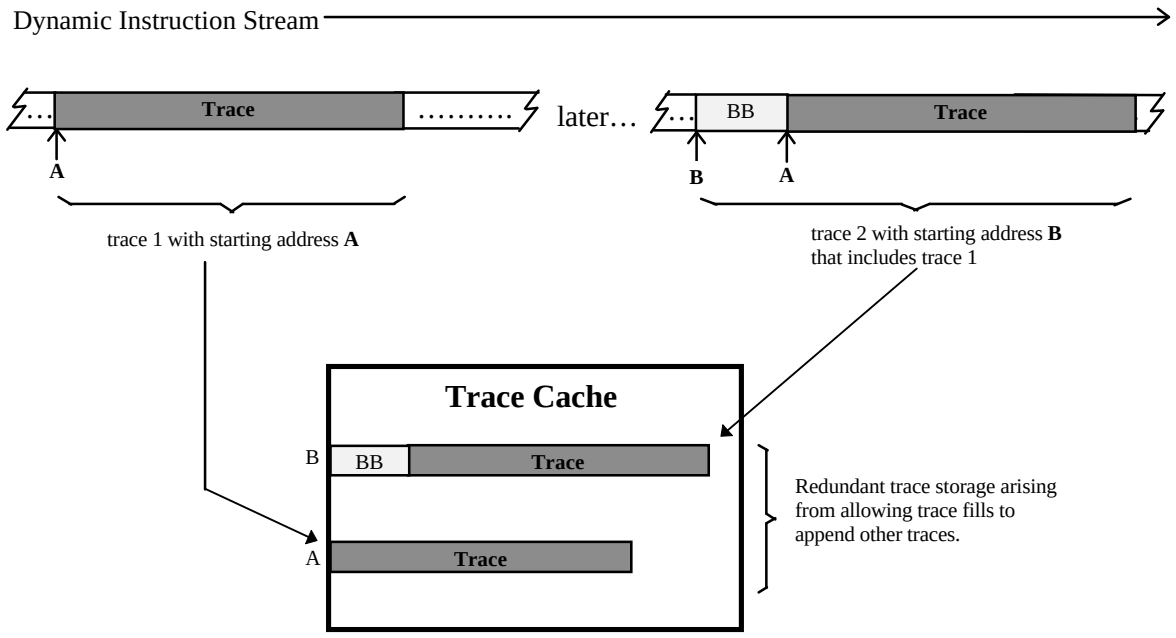

Figure 6-17: Redundant Traces from Trace Fills Allowing Trace Hits

By having trace fills abort on a trace hit, the trace cache can achieve a higher level of storage efficiency. However, based on simulation results, the fetch IPC does not improve much at all; in fact, fetch performance actually decreases for a majority of the benchmarks. Figure 6-18 shows the simulation results, as compared with the base trace cache design:

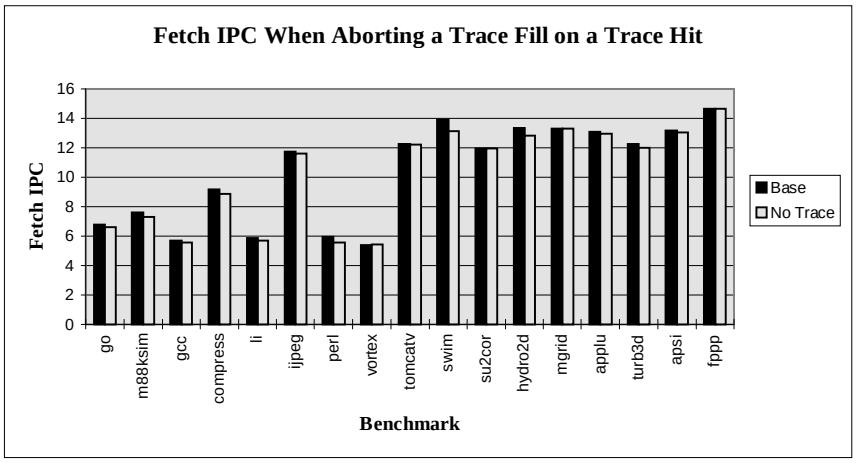

Figure 6-18: Fetch IPC When Aborting a Trace Fill on a Trace Hit

It appears that since trace hits occur relatively frequently, the no-trace requirement results in many truncated trace fills. This requirement causes much smaller traces to be written into the trace cache, which is ultimately detrimental to fetch IPC. Apparently, the reduced trace redundancy from not allowing traces in a trace fill is outweighed by the truncation problem. Ironically, this negative effect on fetch IPC becomes more pronounced as the trace cache hit rate increases. For example, for larger sized trace caches, overall fetch IPC actually decreases as the probability of truncated traces becomes larger and overrides the benefits of having a larger cache.

Although the fetch IPC suffers from this fill mechanism modification, there is actually a decrease in the trace cache miss rates, as shown in Figure 6-19:

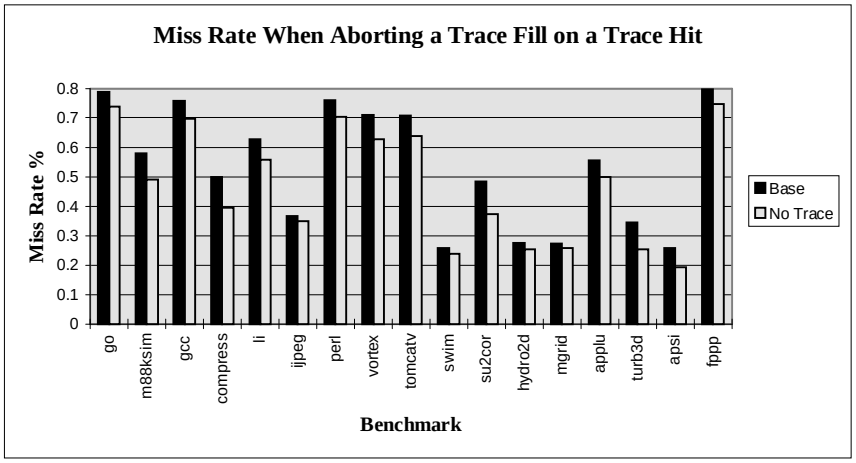

Figure 6-19: Miss Rate When Aborting a Trace Fill on a Trace Hit

Thus, although fetch IPC does not improve, not including traces in a trace fill results in better trace cache miss statistics. This may be because there is a higher probability that a smaller trace resulting in a trace cache hit, since it depends less on branch outcomes than longer traces with more basic blocks. This interesting result presents a tradeoff of using this trace fill modification. One may want to use trace fills that do not include exiting traces depending on whether the goal is to increase fetch IPC or to decrease miss rates.

6.6.3 Specifying End Branch Direction

In the original trace cache design, a target address as well as a fall-through address is designated. This may cause unnecessary overhead in the case where a branch does not terminate the trace, in which case the target address and fall-through address are set as the same thing. An alternative to storing both the target address and the fall-through address is to just add a third branch flag and do a prediction comparison on the third branch flag as well. Only one target address is specified, based on how the direction of the original branch that ended the trace. Although this reduces the flexibility of using the trace (for example, a trace miss will be generated from the last iteration of a loop where the fall-through case is used), the fetch performance results do not suffer much at all:

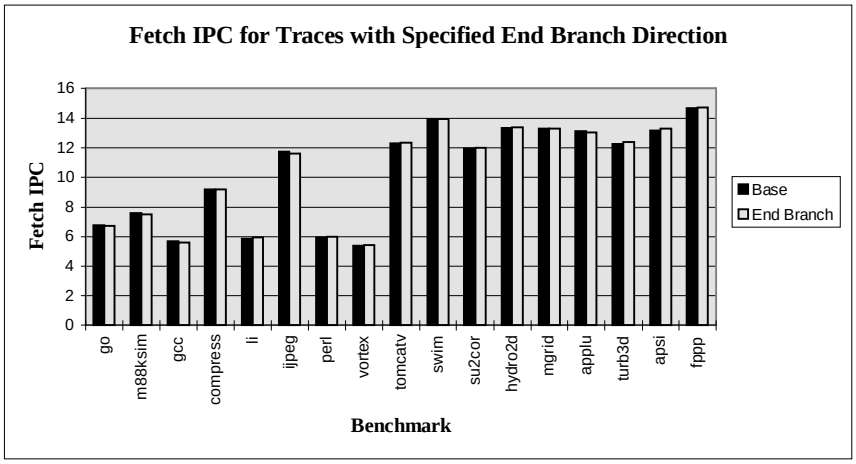

Figure 6-20: Fetch IPC for Traces with Specified End Br. Direction

Based on the results, it is apparent that the benefits of having a fall-through address is limited. This makes sense, since falling out of a loop is not the common case. Thus, specifying the end branch direction in the trace does not affect fetch performance much. Effectively, this saves 4 bytes in storage for every trace line, which can become considerable given the trace cache sizes that maximize fetch IPC (see section 6.9).

6.7 Effects of Partial Hits

One of the prime restrictions with the way traces are stored in the base trace cache design is that the predicted path of execution must match the complete path of a trace. A way of removing this restriction is to allow partial trace hits, where not all the branch predictions have to match in order for there to be a trace cache hit. In this case, if the fetch address matches the trace cache tag, the basic blocks up to the first branch prediction that doesn't match the trace cache branch flags are used, thereby allowing partial traces to be fetched. This is in contrast to the "all or nothing" approach used in the original trace cache design.

In order to support partial hits, intermediary addresses must be stored for each basic block. For every branch that exists within the trace, there is an associated address that indicates where program execution is to continue if the predicted branch direction doesn't match the branch flag. In addition, the trace cache must have some way of determining the end of the associated basic block of a branch. This means that either a pointer for each basic block is needed, or the read logic has a way of identifying the presence of a branch and mask the rest of the instructions. Since it is prohibitively expensive to do an associative search of all the instructions in a trace to determine where the branches are, pointers are stored as part of the trace cache control information. The modifications to allow partial hits entail an additional 8 bytes for intermediate addresses and approximately 1 byte to store pointers.

The trace cache fetch process is modified in the following manner to allow partial hits: the fetch address is again compared with the trace cache tag. If there is a match, then the branch flags are compared with the branch predictions. If there is a mismatch between the prediction and flag, then the associated pointer is used to determine the instructions within the trace that are valid. These instructions are then fed into the instruction latch. The associated intermediary address is then used as the next fetch address. Figure 6-21 shows the modification necessary for the trace cache to support partial hits.

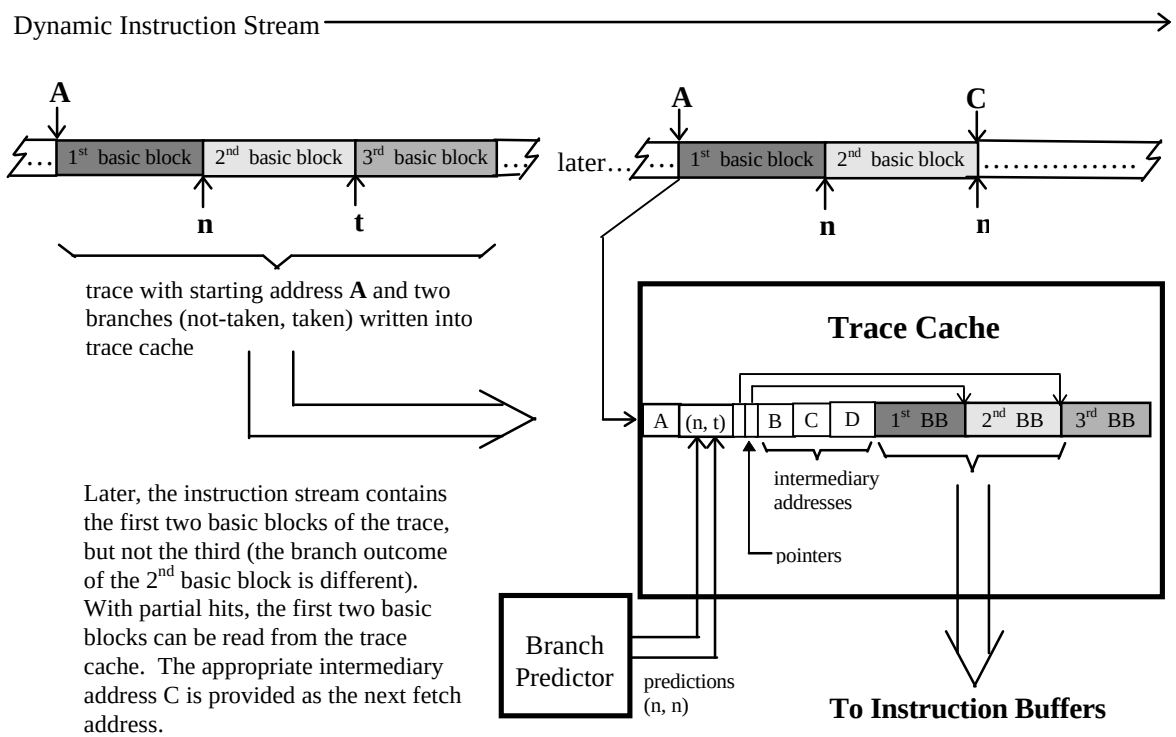

Figure 6-21: Trace Cache Modifications to Allow Partial Hits

By allowing partial hits, we relax the strict requirement that a complete trace must match the predicted program execution path. This dramatically improves trace cache performance, as shown in Figure 6-22. The simulation results show the fetch IPC for the base trace cache design with only the partial hit modification.

The integer benchmarks show an average 9.4% improvement from allowing partial hits, justifying the additional storage overhead needed to support partial hits. An important fact to realize is that now at the very least, one basic block will be read from the trace cache as long as the trace cache tag match occurs. Thus, in the worst case scenario, a trace cache can act as an instruction cache.

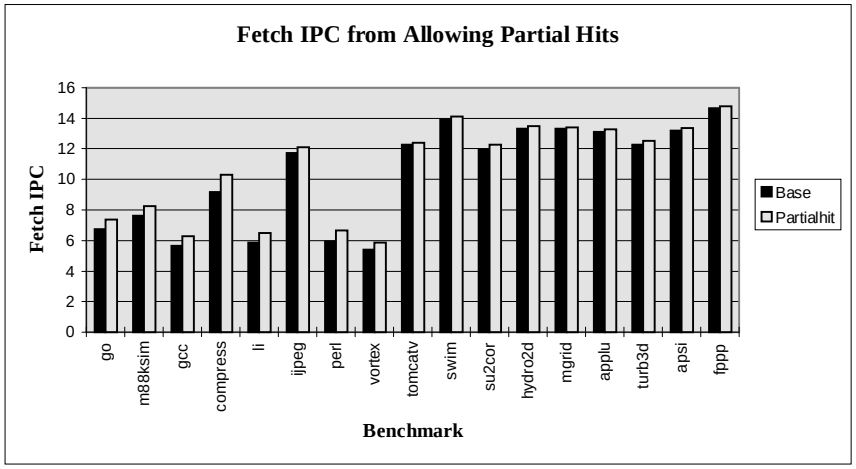

Figure 6-22: Fetch IPC of Trace Cache Allowing Partial Hits

Allowing partial hits also results in much lower trace cache miss rates, as Figure 6-23 indicates:

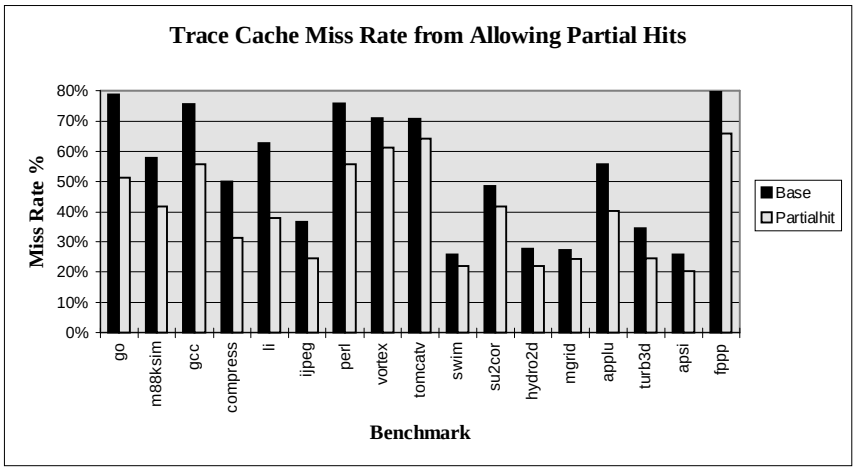

Figure 6-23: Trace Cache Miss Rate from Allowing Partial Hits

The decreased trace cache miss rates can be attributed to the fact that branch prediction mismatches do not cause trace misses anymore. In fact, after initial cold start misses, the only way that a trace cache will miss is because of a tag mismatch. Thus, in a way, allowing partial hits in a trace provides a degree of path associativity, where multiple paths can exist within the trace cache. In this case, paths with common starts can simultaneously exist in trace cache. Another way al-

lowing multiple paths from the same start address is to simply provide associativity in the trace cache, as shown in the next section.

6.8 Effects of Associativity

Another simple method of providing path associativity and preventing collisions within the trace cache is to increase the associativity of the trace cache. Associativity effectively provides a form of multiple path selection whereby different traces based on the same start address can be stored simultaneously in the trace cache. Thus, associativity allows the trace cache to store complete alternative traces with the same start address, or completely different traces that happen to be mapped to the same line in the trace cache. This is in contrast to partial hits, which can only provide the prefix of a trace if another path of execution with the same start address root is chosen. Figure 6-24 show the integer fetch IPC results for a 4 Kbyte set-associative trace cache corresponding to a 64-entry direct mapped trace cache:

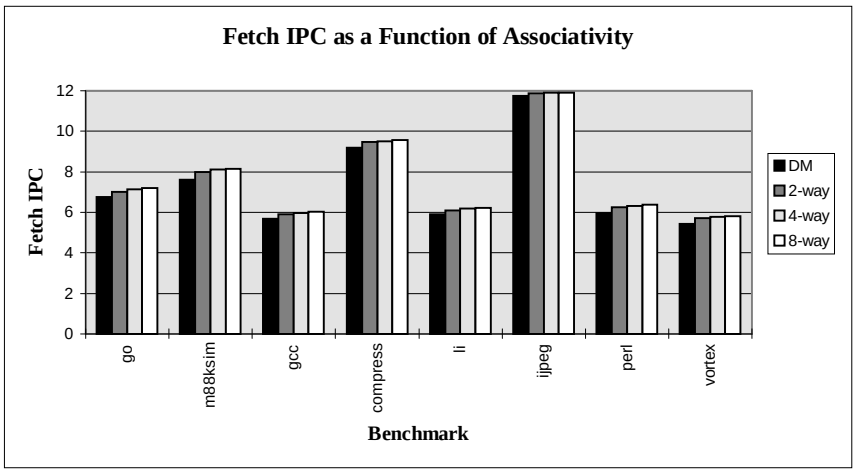

Figure 6-24: Integer Fetch IPC from Increased Cache Associativity

From Figure 6-24, we see a slight improvement in the fetch IPC provided by associativity, with diminishing returns after 2-way associativity. Because of the modest gains in fetch improvement, we can conclude that the majority of the misses within the trace cache are capacity misses rather than conflict misses. This makes design decisions easy because increasing the associativity results

in increased access time and replacement logic complexity and is not economical beyond a certain size. However, providing a small degree of associativity such as 2-way is justifiable for improving performance. It must be noted that these results are based on small trace cache configurations. It remains to be seen what the effects of associativity are for larger sized trace caches.

We can see how the associativity improves trace cache performance from decreasing address collisions by measuring the percentage of trace cache misses that arise from starting tag mismatches. For a 2-way set associative trace cache, the trace cache miss breakdown is as follows:

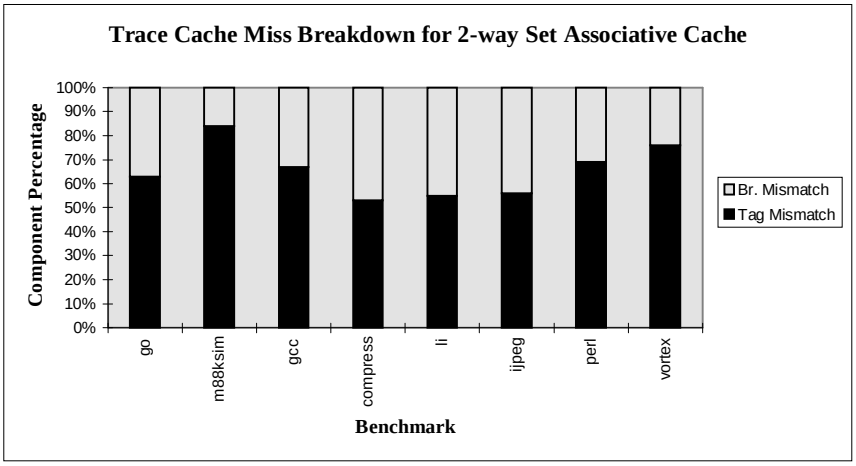

Figure 6-25: Trace Cache Miss Cause Breakdown for a 2-way Set Associative Trace Cache

From Figure 6-25, we see that by providing associativity over a direct-mapped cache, there is a significant decrease in the trace cache misses from a tag mismatch. This reduction indicates that associativity is very effective in reducing the number of misses due to strict address collisions as well as replaced traces with the same start address.

6.9 Effects of Trace Cache Size

Thus far, all the results have been presented for a small 64-entry trace cache. For such a small number of entries, not many useful traces can be stored in the trace cache. Also, there is a much greater probability of address collisions, resulting in possibly useful traces being displaced from the cache. Trace cache performance can be improved dramatically by increasing the trace cache

size. This section attempts to explore how trace cache performance scales with increased trace cache size.

For these simulations, the best fill mechanism that maximizes performance is used (storing complete basic blocks, allowing traces in fills, specifying the end branch direction, and partial hits). The trace cache size is doubled from between 64 entries to 4096 entries. The results for the integer benchmarks are shown in Figure 6-26:

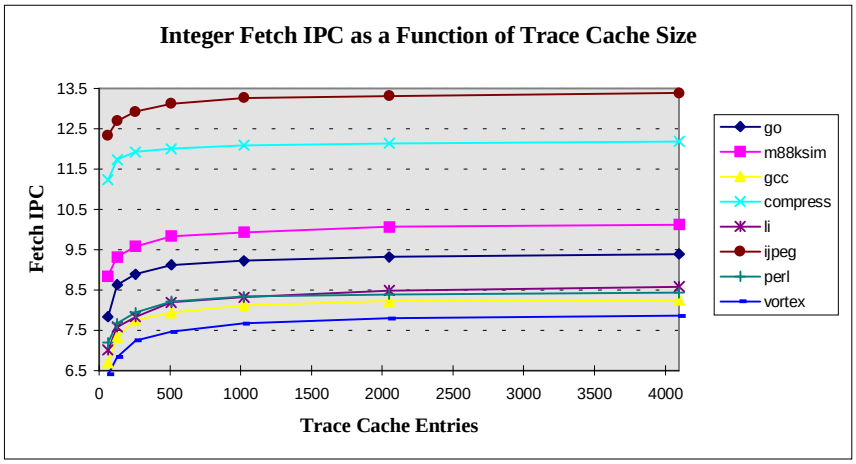

Figure 6-26: Integer Fetch IPC as a Function of Trace Cache Size

and for the floating point benchmarks:

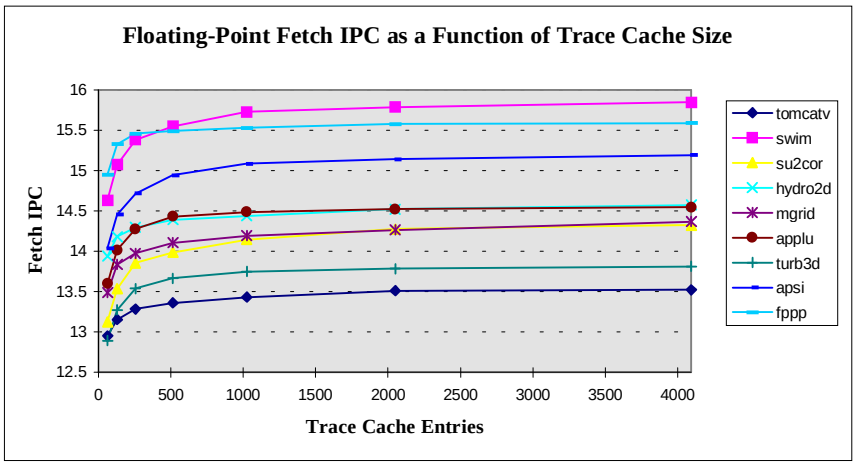

Figure 6-27: FP Fetch IPC as a Function of Trace Cache Size

From the graphs, we see that increasing the number of trace cache entries has a great impact on the fetch IPC performance. We see that in general, fetch performance increases significantly as the trace cache is doubled up to 512 entries. However, beyond 1024 entries, corresponding to a 64K trace cache, fetch IPC improvement drops dramatically. Although the fetch IPC does not completely level off as the trace cache size after 1024 entries, this improvement does not justify the associated costs of doubling the storage space. The fact that fetch IPC improves even for very large trace caches indicates that collisions within the trace cache is always a factor, unless the complete program can be stored in the trace cache. Looking at the trace cache miss rates, we can see that increasing the number of entries has a dramatic effect on trace cache efficiency as well:

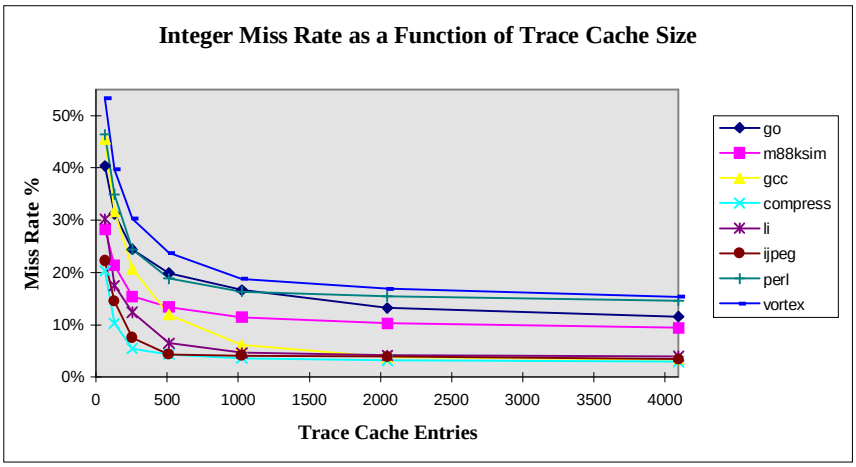

Figure 6-28: Integer Miss Rate as a Function of Trace Cache Size

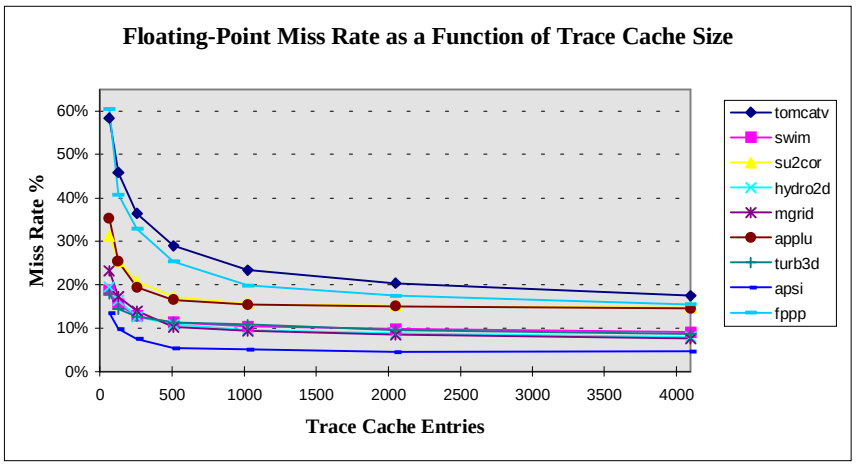

Figure 6-29: FP Miss Rate as a Function of Trace Cache Size

The miss rates drop sharply as the cache size is increased from 64 entries, indicating that at that size, there are a great deal of collisions with the trace cache. From the figures, it appears that the trace cache miss rates level off after 1024-entries. For a 1024-entry trace cache, compress, li and ijpeg reach a remarkably low miss rate of around 4%. The integer benchmark gcc improves the most, dropping from 45% to a mere 6%. In general, the trace cache miss rates drop to around 10% for integer benchmarks, and 13% for floating-point benchmarks. Again, although these miss rates are much lower than for the original 64-entry trace cache, they are still not comparable to the miss rates that can be found for standard instruction caches.

6.10 Replacement policy

Another basic specification of a cache is the replacement strategy when deciding what elements of the cache are to be replaced. In the associativity section, the trace cache replacement policy was implemented as a random selection from the trace cache entries. There are other alternatives that could possibly improve the trace cache performance. The most common policies include Least Recently Used (LRU) and simple Round Robin replacement.

A comparison of the different replacement policies on a 1024-entry, 2-way set associative trace cache are shown in Figure 6-30:

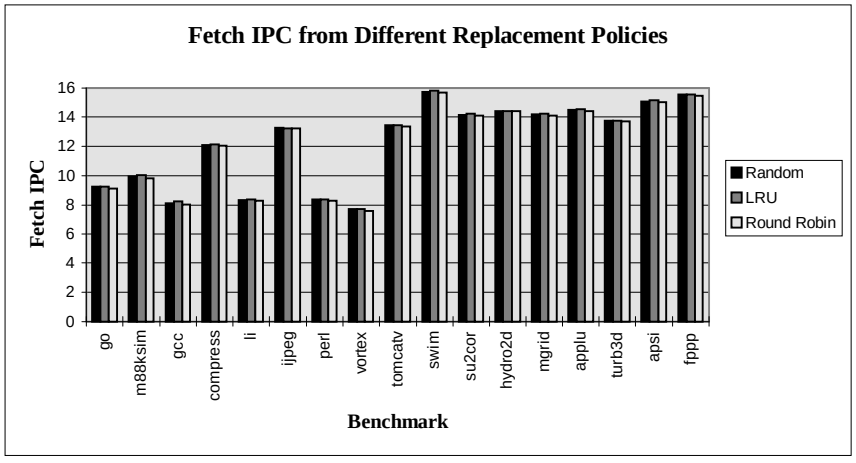

Figure 6-30: Fetch IPC from Different Replacement Policies

In general, the LRU replacement algorithm performs the best, followed by random replacement. However, there is a negligible difference in performance between the three replacement strategies. This is probably due to the large size of the trace cache that the replacement strategy was tested on. The results seem to indicate that it makes little difference what replacement strategy is chosen for the trace cache. Whatever is easiest or least expensive to implement should be the chosen algorithm.

6.11 Effects of Real Branch Prediction

All of the results presented above have assumed the presence of a perfect branch prediction mechanism in order to isolate the performance of the trace cache. As such, the results represent the maximum achievable performance by the trace cache. Since the performance of the branch predictor has a direct impact the trace cache performance, this section provides trace cache statistics using a real branch predictor.

For the branch prediction simulations, the correlated multiple branch predictor discussed in section 3.2 is used. The simulations assume a 16 Kbyte PHT with a 16-bit global history register. Figure 6-31 shows the results of using real branch predicton:

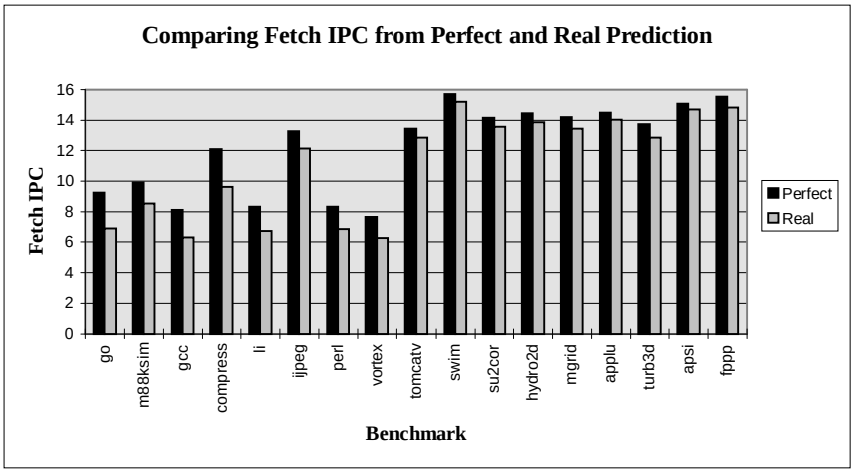

Figure 6-31: Fetch IPC from Using Real Branch Predictor

The fetch IPC results from using real branch prediction are much smaller than the perfect branch prediction values. The cause of the drop in fetch IPC is the branch mispredictions from the real branch predictor. These mispredictions necessitate that the corresponding mispredicted traces be aborted, which causes instructions to be flushed from the pipeline. The disparity between the perfect and real branch prediction IPC values indicates that the potential of the trace cache to increase fetch IPC is highly dependent on an accuracy of the branch predictor. From the Figure 6-31, we see that the integer benchmarks suffer much more from using a real branch predictor. This can be understood from Figure 6-32, which shows the branch misprediction rates for the different benchmarks:

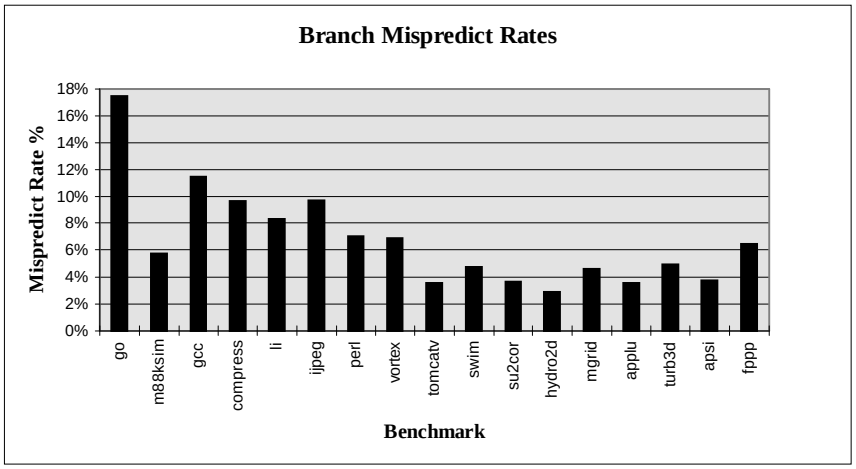

Figure 6-32: Branch Misprediction Rates

As we can see, the branch mispredict rates are much higher for the integer benchmarks than for the floating-point benchmarks, with an average of 9.5% and 4.3% respectively. This makes sense, because the floating-point benchmarks generally have highly predictable loops and exhibit fewer complicated branch structures.

These misprediction rates seem respectively low, but represent only single branch mispredictions. Given a prediction accuracy of p (where $0 \leq p \leq 1$) and a trace that contains three branches, the chances that the trace will be predicted correctly drops to p^3. For example, given a single branch prediction accuracy of 90%, the chances that a trace contains three branches will be predicted cor-

rectly is only 73%. This example clearly illustrates the absolute need for high branch prediction accuracy in order to make trace caches effective.

Chapter 7

Conclusions

There has been a trend toward increased superscalarity and aggressive speculation techniques to extract instruction level parallelism in modern microprocessors. As such, much more aggressive instruction fetch mechanisms are needed to be able to fetch multiple basic blocks in a given cycle. One such fetch mechanism that shows a great deal of promise is the trace cache, originally proposed by Rotenburg, et. al. In this thesis, a number of critical design issues regarding the trace cache fetch mechanism were explored.

The results indicate that the fill mechanism for the trace cache can significantly affect overall trace cache performance. One important trace cache optimization involves storing complete basic blocks for a trace instead of being able to truncate a trace arbitrarily. By storing only complete basic blocks, redundancy within the trace cache is significantly reduced, resulting in an average 6.4% improvement in the fetch IPC and a 17.2% decrease in the trace cache miss rate for integer benchmarks.

Another important conclusion, perhaps the most significant, is the performance benefits of supporting partial hits within traces. By relaxing the trace cache hit requirement, the trace cache can be made to act effectively as an instruction cache in the worst case, with the potential of providing more basic blocks depending on the branch predictions. This decreases the trace cache miss rate by 29.8% and 18.3% for integer and floating-point benchmarks respectively. These trace cache miss rate reductions result in an average integer fetch IPC improvement of 9.4% for the smallest trace cache configuration.

This research also demonstrates the dependency of the trace cache on the branch predictor accuracy. Since the primary motivation behind the trace cache is to be able to fetch multiple basic blocks every cycle, an accurate multiple branch predictor is of paramount importance. The branch

predictor presented by Rotenburg, has an average prediction accuracy of 90.5% for integer benchmarks and 95.7% for floating-point benchmarks. Although these prediction rates are very admirable, there is still a great deal of room for improvement. Multiple branch prediction performance is becoming an important research area. The performance of the trace cache will undoubtedly increase as techniques to enhance multiple branch predictor performance continue to be developed [25]-[27].

Finally, the simulation results point out the important fact that size does matter. By increasing the trace cache from the original 4Kbyte size to a 64Kbyte configuration, there is an average fetch IPC improvement of 15.2% for integer benchmarks, and 5.8% improvement for floating-point benchmarks. This 64Kbyte configuration is on par with the size of the primary instruction cache (128 Kbytes), indicating that a respectably-sized trace cache can considerably improve fetch performance.

One idea that was mentioned in the course of this thesis was the possibility of having the trace cache completely replace the primary instruction cache as the main fetch mechanism. In this scenario, the L2 cache would be used as a fetch mechanism for the trace cache. However, this idea requires that the trace cache hit rate be very high in order to be effective. Based on the simulation results of this research, such a fetch mechanism is not feasible; the miss rate of trace cache is inherently higher than instruction cache (average of 10.2% and 13.3% for integer and floating point benchmarks respectively).

With all of beneficial design attributes included, an optimized trace cache design performs an average of 34.9% better for integer benchmarks, and 11.0% better for floating-point benchmarks than the originally proposed trace cache design. This corresponds to a 67.9% and 16.3% improvement in fetch bandwidth over a traditional instruction cache, for integer and floating-point benchmarks respectively. The results presented demonstrate that the concept of a trace cache is still very nascent and open to innovation. More studies like the one done in this thesis are warranted to further develop the viability of the trace cache as a high performance fetch mechanism.

Chapter 8

Future work

The concept of the trace cache is gaining support as a feasible fetch mechanism to increase fetch bandwidth. In fact, several groups have proposed new processor architectures based on the trace cache [13], [14]. Although this research addresses a number of important facets of trace cache design, there is still a great deal of research that can be done, including exploration of the following topics:

- ***Victim Caches***: A well-known concept for caches, whereby a replaced trace from a cache can be stored in a small buffer instead of being thrown out completely. This is to reduce the possibility that an address collision might cause a useful trace from being displaced permanently from the trace cache.
- ***Judicious Trace Selection****:* The current trace cache implementation simply stores all completed traces from a trace miss fill. More judicious trace selection algorithms might improve performance. One example is creating a commit buffer that commits a completed trace fill to the trace cache only if a hit to that trace occurs. Another idea is to only commit a trace if it is longer than the existing trace that it is to replace.
- ***Trace Storage***: There are many issues of how to store traces within the trace cache. The simplest method is to simply allocate a specified trace length as the cache block size, truncating traces that are too long and leaving empty space with traces that are too short. Alternatively, there may be more ambitious indexing methods that can increase storage efficiency, at the cost of complexity as a result of non-uniform trace sizes.
- ***Path Associativity***: In this research, only standard associativity is considered. With standard associativity, if multiple traces within a set match the fetch address, the first trace is used. An optimization is to compare all trace hits and provide the *longest* trace to the instruction latch.

- ***Fill Issues***: The fill mechanism assumed in this research simply serialized trace fills, only allowing one fill at a time and ignoring new trace misses until a trace is completely written. There are alternative fill mechanism designs, such as delaying servicing new misses until after a trace fill in progress is completed or allowing for concurrent fills. Also, in the trace cache implementation used in this research, a trace fill automatically truncates when a return instruction is encountered. One can allow returns in a trace if the next fetch address is provided by the return address stack (RAS) instead of the trace cache.

- ***Non-speculative Trace Fills:*** In the current implementation of the trace cache, trace fills are speculative in the sense that they do not wait for branch outcomes before committing the trace to the trace cache. An alternative is to do the trace fill from graduated instructions, so only traces from the real path of execution are stored.

- ***Indexing***: The only form of indexing addressed in this thesis was the gshare indexing algorithm. There are other indexing schemes based on concatenation of fetch address and branch prediction bits or global history that may be more effective.

- ***Branch Prediction***: As we saw in section 6.11, branch prediction accuracy affects fetch performance of the trace cache tremendously. There are other possible multiple branch prediction schemes, many of which are based on existing branch predicting techniques that may be incorporated to improve prediction accuracy [16] - [27].

Many of the above ideas complicate the trace cache control logic and may increase access latency. In addition, multiple design choices can interact in complicated and subtle ways. Detailed simulation studies are needed to determine their relative performance benefits. In addition, more thorough analysis of how increased instruction fetch bandwidth affects overall processor performance is warranted.

Bibliography

[1] E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996

[2] T-Y Yeh, D. Marr, and Y. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993

[3] S. Dutta and M. Franklin, "Control Flow Prediction With Tree-like Subgraphs for Superscalar Processors," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995

[4] T. Conte, K. Menezes, P. Mills, and B. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates," *Proceedings of the International Symposium on Computer Architecture*, June 1995

[5] S. Melvin, M. Shebanow, and Y. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proceedings of the 21st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998

[6] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue," *Proceedings of the 27th Annual International Symposium on Microarchitecture*, December 1994

[7] M. Smotherman and M. Franklin, "Improving CISC Instruction Decoding Performance Using a Fill Unit," *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, 1995

[8] S. Wallace and N. Bagherzadeh, "Multiple Branch and Block Prediction," *Proceedings of the 27th Annual ACM/IEEE Conference on High Performance Computer Architecture*, 1997

[9] Q. Jacobsen, E. Rotenburg, and J.E. Smith, "Path-based Next Trace Prediction," *Proceedings of the 30th Annual Symposium on Microarchitecture*, December 1997

[10] K. Menezes, S. Sathaye, and T. Conte, "Path Prediction for High Issue-Rate Processors," *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques, November 1997.*

[11] M. Butler, T-Y Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Instruction Level Parallelism is Greater Than Two," *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991

[12] E. Hao, P-Y Chang, M. Evers, and Y. Patt, "Increasing the Instruction Fetch Rate via Block-Structured Instruction Set Architectures," *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996

[13] S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997

[14] E. Rotenburg, Q. Jacobsen, Y. Sazeides, and J.E. Smith, "Trace Processors," *Proceedings of the 30th Annual Symposium on Microarchitecture*, December 1997

[15] T-Y Yeh and Y. Patt, "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution," *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992

[16] J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981

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

[18] S. McFarling, "Combining Branch Predictors," *Technical Report TN-36*, Digital Western Research Laboratory, June 1993

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

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

[21] T-Y Yeh, " Two-level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of Michigan, 1993

[22] T-Y 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

[23] E. Hao, P. Chang, Y. Patt, "The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited," *Proceedings of the 27th Annual International Symposium on Microarchitecture*, December 1994

[24] B. Calder and D. Grunwald, "Fast & Accurate Instruction Fetch and Branch Prediction," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994

[25] C. Young, N. Gloy, and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," *Proceedings from the International Symposium of Computer Architecture*, June 1995

[26] P-Y Chang, M. Evers, and Y. Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," *Proceedings of the 1996 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1996

[27] P-Y Chang, E. Hao, Y-Y Yeh, and Y. Patt, "Branch Classification: A New Mechanism for Improving Branch Predictor Performance," *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, 1994

[28] L. Gwennap, "MIPS R1000 Uses Decoupled Architecture," *Microprocessor Report*, Oct. 1994

[29] A. Agarwal, "UltraSPARC: A New Era in SPARC Performance," *Proceedings of the 1994 Microprocessor Forum*, Oct. 1994

[30] M. Slater, "AMD's K5 Designed to Outrun Pentium," *Microprocessor Report*, Oct. 1994

[31] S. McFarling, "Program Optimization for Instruction Caches," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989

[32] W. W. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proceedings of the 16th Annual International Symposium of Computer Architecture*, May 1989

[33] C. L. Mitchell and M. J. Flynn, "The Effects of Processor Architecture on Instruction Memory Traffic," *ACM Transactions for Computer Systems*, August 1990

[34] R. Gupta and C.-H. Chi, "Improving Instruction Cache Behavior by Reducing Cache Pollution," *Proceedings of the 1990 Conference on Supercomputing*, November 1990

[35] D.B. Whalley, "Fast Instruction Cache Performance Evaluation Using Compile-time Analysis," *Proceedings of the ACM SIGMETRICS 1992 Conference on Measurement and Modeling of Computer Systems*, June 1992

[36] D. B. Whalley, "Techniques for Fast Instruction Cache Performance Evaluation," *Software Practice and Experience*, January 1993

[37] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems, October 1987*

[38] W. Hwu, et. al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, January 1993

[39] D. Stiliadi and A. Varma, "Selective Victim Caching: A Method to Improve the Performance of Direct Mapped Caches," *Proceedings of the 27th Hawaii International Symposium on System Science*, January 1994

[40] S. J. Walsh and J. A. Board, "Pollution Control Caching," *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, 1995

[41] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Second Edition. Morgan Kaufmann, 1996