HIGH-PERFORMANCE FRONTENDS FOR TRACE PROCESSORS

by

Quinn Able Jacobson

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN – MADISON

1999

ABSTRACT

Today's microprocessors achieve instruction-level parallelism by aggressive pipelining and the ability to process multiple instructions in parallel. A typical processor can be broken into two main parts, the frontend and the backend. The frontend sequences through the static representation of the program and creates a dynamic stream of instructions to feed the backend. The backend executes the instructions.

Trace processors make use of a new microarchitecture organization that enables them to achieve higher throughput in both the frontend and backend. In the frontend a trace processor uses a trace cache to enable it to fetch across multiple branches in a single cycle. The trace cache records short dynamic sequences of instructions, traces, and can provide one trace of instructions per cycle when a path is repeated. Traces are dispatched, one per processing element, to a distributed backend. The backend is constructed of simple execution engines and high aggregate throughput is achieved through replication.

This thesis proposes three mechanisms that enable a very high-performance frontend for trace processors. The first mechanism, trace pre-construction, augments the trace cache by performing a task analogous to prefetching. It increases both the average performance of the trace cache, and the robustness of the trace cache to varying workloads. Pre-construction can reduce the trace cache miss rates up to 80%.

The second mechanism, instruction pre-processing, takes advantage of the trace cache to dynamically optimize applications. It can enable transformations that both dynamically optimize common instruction sequences, and take advantage of implementation-specific hardware. The dynamic optimizations, performed in the frontend, can expose more parallelism to the trace processor backend. Three specific optimizations are considered: instruction scheduling, constant propagation and instruction collapsing. Together these optimizations increase performance by up to 20%.

The third mechanism, next-trace prediction, is a control predictor that can match the bandwidth of the trace cache without sacrificing accuracy. It performs the functionality of branch prediction and branch target prediction. It works in units of traces, so its bandwidth is perfectly matched to the trace cache. Next-trace prediction has prediction accuracy comparable to the best traditional branch predictors, while providing significantly higher branch throughput.

Together with the trace cache, these mechanisms produce a high-performance frontend that offers high instruction fetch bandwidth and exposes more parallelism to the backend.

ACKNOWLEDGMENTS

Of all the parts of this thesis, I have found this one the hardest to write. There are so many people that have played special roles in my life. This thesis is a testament to their support and understanding over the years. There is not space to list everyone who deserves to be acknowledged.

First and foremost I would like to thank my family, my loving wife Lindsay and our son Ezra. The support and love of my wife has been the most valuable help I could ever hope to receive. This work is as much a testament to her support and sacrifices as to my own achievements. I would like to also thank Ezra for his love and support. From him I have learned the meaning of unconditional love and the absolute wonder of discovery.

I would like to thank all the faculty of the ECE and CS departments at the University of Wisconsin that I had the pleasure of working with and learning from. I would like to thank them for sharing their experience and knowledge, and also for treating me as a colleague. They not only taught me more about computers, but also the meaning of professionalism. I would like to especially thank James E. Smith and Pei Cao with whom I had the honor of writing papers. I would also like to thank the outstanding graduate student body of the University for establishing and maintaining a tradition of excellence. The students that have come before me have set a high standard that I hope I have lived up to. I will leave the University of Wisconsin proud to say that I was part of something special.

I would especially like to thank my advisor, James E. Smith, for being my mentor and friend. He has kept me on track while providing the freedom to find my own way. His guidance has been invaluable. I know that I am a better person for having worked with him.

Professor Gurindar Sohi has been integral part of my experience at Wisconsin. He has been a mentor and a role model. His high standards rub off on those around him and better the entire field of computer architecture.

I would like to thank the professors who served on my preliminary and final defense committees: James E. Smith, Charles Kime and Kewal Saluja from ECE and Gurindar Sohi and David Wood from CS. Their guidance has been very important. I would also like to thank professor Mark Hill for serving on my final defense committee.

There have been many important teachers in my life. From UC Santa Cruz, I would like to thank Professors Anujan Varma and Glen Langdon. From Long Beach Polytechnic High School I would like to thank Jackie Deamer and Anne Maben.

The support of my parents has made this work possible. Their value of education and exploration lead me to where I am today. Their belief in me gave me the confidence in myself that keeps me going. They have been with me every step of the way, with both emotional and material support. My sisters, Willow and Brynna, have also been an important part of my life.

Friendship is the most important part of life (work is only a distant second). To all my friends, I would like to say thank you for everything. To save space (and to avoid the danger

of forgetting someone) I will not attempt to make a list. You know who you are, and my thoughts are with you.

For their direct help in producing this thesis I would like to thank James E. Smith, Eric Rotenberg and Todd Bezenek. Jim's help in performing the research and his help in preparing this document has been invaluable. I would like to thank Eric for his insights and for his help in developing the simulation infrastructure that made this work possible. I would like to thank Todd for his help of proofreading this document.

I would like to acknowledge the agencies that provided the financial and equipment support that made this work possible. Intel Corporation supported part of my graduate work through a fellowship and through the Intel Equipment Grant to the UW donated the computational resources that were used to perform much of the work in this thesis. The NSF partially supported this work through Grant MIP-9505853. The U.S. Army Intelligence Center and Fort Huachuca partially supported this work under Contract DAPT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

-Quinn A. Jacobson
August, 1999

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

The trend toward ever-faster microprocessors must be continued to meet tomorrow's computing needs. Faster, denser chip technologies help continue performance growth, but new microarchitectures are needed to push performance further and to use higher transistor counts effectively. We are quickly reaching the point where a fundamental new processor organization, beyond conventional superscalar, is needed. A possible candidate is the trace processor organization [65][57][49]. Along with a new processor organization come many opportunities for novel microarchitectural mechanisms to address new opportunities and solve weaknesses that arise. This thesis focuses on a set of new microarchitectural mechanisms to increase the performance of trace processors.

1.1 Instruction-Level Parallelism Processors

Over the past few decades many ingenious approaches have been proposed to increase processor performance. Increased performance comes from both increasing clock speeds and increasing the amount of work done per clock cycle. Clock speeds are largely dictated by circuit technology, which has been advancing at a steady exponential rate [52]. Processors have been able to maintain a rate of performance improvement above circuit technology

growth by increasing the amount of work done per cycle, exploiting what is called instruction-level parallelism (ILP).

In a typical processor, instruction-level parallelism is exploited in two ways. First, parallelism is exploited by pipelining [4], where different phases of instruction processing are performed in an overlapped fashion. Second, parallelism is exploited by superscalar execution [59], where multiple instructions are issued and executed in parallel. Much of the potential benefit of the former has already been exploited in previous generations of microarchitectures, and future designs are increasingly focused on exploiting the benefit of the latter.

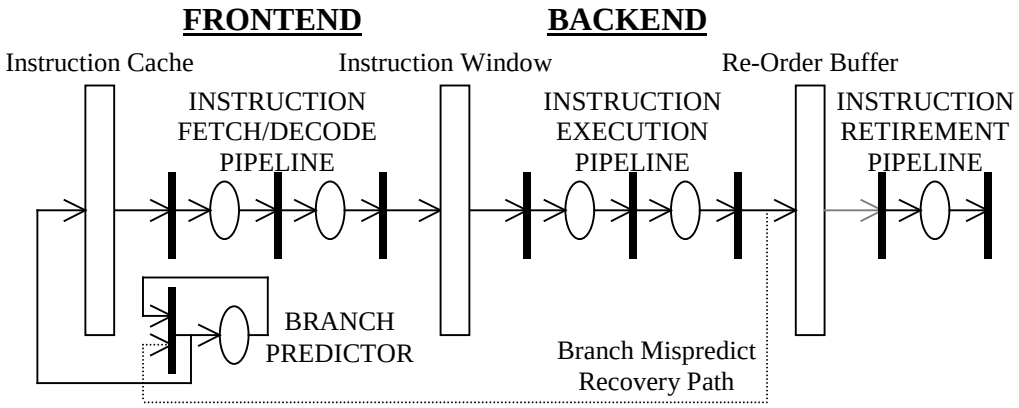


Figure 1-1 Typical processor organization.

A typical high-performance processor can be broken into two main parts, the frontend and the backend (see Figure 1-1). The processor's frontend sequences through a static representation of a program (the binary) and creates a dynamic stream of instructions to feed the backend. The frontend consists of a branch predictor, that primarily directs sequencing, and a decode pipeline to decode and analyze instructions. The frontend processes

instructions in program order, although it may have a throughput of multiple instructions per cycle.

The processor's backend executes the instruction stream and provides feedback to the frontend about control flow directives in the program. The backend contains an execution pipeline that can be decoupled from the frontend with an instruction window. The execution pipeline may take instructions in an out-of-order fashion [60] from the instruction window, obeying only true dependencies, and may also have a throughput of multiple instructions per cycle. In addition to the execution pipeline, the backend has a retirement pipeline that commits the work performed by instructions. For an out-of-order processor the retirement pipeline is decoupled from the execution pipeline by a re-order buffer [56]. The retirement pipeline is in-order.

The performance of the processor's frontend is limited by branch instructions, instructions that conditionally or unconditionally redirect the processor to fetch from another location. In order to keep the pipeline fed, the frontend must predict the behavior of branch instructions (their target and whether they are taken or not), as the instructions will not be decoded and executed until later in the pipeline. Today's processors can only predict one branch instruction per cycle (branch instructions can occur as often as one out of four instructions for some applications). Even if multiple branch instructions were predicted, a normal instruction cache can only fetch a contiguous run of instructions up to the first taken branch. When the prediction for a branch is wrong, the instructions following the branch are flushed from the pipeline and the processor starts refilling the pipeline with the correct

instructions. The penalty for mispredictions, in terms of lost potential work, can be very large.

The performance of the processor's backend depends on the execution resources available and the amount of parallelism exposed in the instruction stream. Adding more execution resources increases the peak throughput of the backend. Increasing the size of the instruction window, if the frontend can keep it full, increases the number of instructions exposed to the backend. The more instructions the backend can observe, the more likely it can find independent instructions to execute in parallel. The ability to increase the number of execution resources and the size of the instruction window is limited by size constraints and logic complexity. The more execution resources or the larger the instruction window, the more complex and potentially slower the necessary logic will be.

A processor requires a peak instruction fetch bandwidth considerably higher than the sustained instruction throughput of the processor. The higher bandwidth is needed because some fraction of the fetch bandwidth is wasted fetching instructions that correspond to incorrect speculative paths. More importantly, the high bandwidth is needed to refill the instruction window quickly (to expose instructions to the backend) after branch mispredictions.

There are a number of factors limiting the scalability of the general superscalar microarchitecture of today's high-performance processors. Today's processors achieve an average throughput in the range of one to two instructions per cycle. To sustain a higher throughput will require the ability to fetch and decode more than one basic block per cycle.

An entirely new approach to instruction fetch is required to achieve the needed fetch bandwidth. Significantly higher peak throughputs are also needed in the backend to sustain higher average throughputs. To achieve this requires overcoming complexity constraints that limit the size of the instruction window and the number of execution resources that can be incorporated.

1.2 Overview of the Trace Processor microarchitecture

The trace processor microarchitecture has the potential to enable higher amounts of instruction-level parallelism than can be achieved with today's superscalar microarchitectures. The trace processor microarchitecture addresses the factors limiting the throughput of both the frontend and the backend of processors. This enables the development of processors that can achieve higher performance while maintaining complete compatibility with existing instruction sets.

I use the term "trace processor" to refer to a processor built around the concept of a trace cache [41][48][40] and organized for the execution of traces (for both the processor frontend and backend). Specifically I refer to the microarchitecture organization proposed by Vijapeyam and Mitra [65] and by Rotenberg, Sazeides, Smith and myself [49]. This microarchitecture organization built on a number of other works [14][58][48][57][3]. In this section the general concept and motivation of the trace processor paradigm is discussed. A detailed description of the actual trace processor hardware is given in the next chapter.

A *trace* is a valid dynamic sequence of instructions not longer than some maximum length (16 or 32 for example). The instructions in a trace need not be contiguous in the static representation of the program and an instruction can occur multiple times in the same trace and in multiple different traces. Traces are constructed during program execution and stored in a special cache, a *trace cache*. When there is a trace in the cache that matches the expected dynamic behavior of the program, it can be fetched from the trace cache in a single cycle.

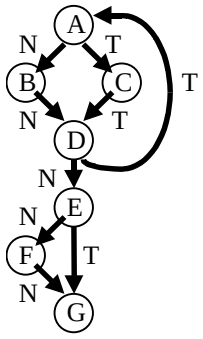


Figure 1-2 Example CFG.

Trace cache operation can best be understood via an example. Figure 1-2 shows a program's control flow graph (CFG), where each node is a basic block, and the arcs represent potential transfers of control. In the figure, arcs corresponding to branches are labeled to indicate taken (T) and not taken (N) paths. The sequence *ACD* represents one possible trace, which holds the instructions from the basic blocks *A*, C, and *D*. This would be the sequence of instructions beginning with basic block *A* where the next two branches are taken. These basic blocks are not contiguous in the original program, but would be stored as a contiguous block in the trace cache (see Figure 1-3).

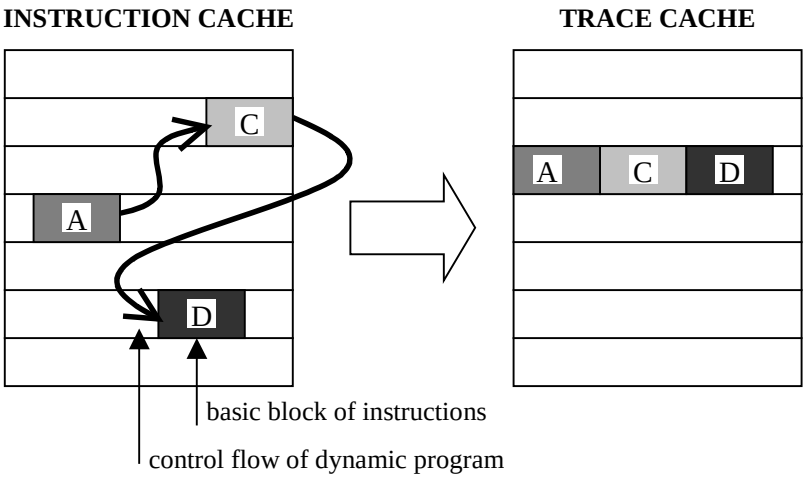


Figure 1-3 Instruction cache and trace cache representations of a program.

The trace cache replaces the instruction cache as the primary source for fetching instructions, providing an entire trace worth of instructions per cycle. To use the increased fetch bandwidth of the trace cache, the branch predictor must also have increased bandwidth. This is achieved with a special predictor that works in units of traces: a next-trace predictor (discussed in detail in Chapter 5). The decode pipeline is also modified to support the higher bandwidth, decoding and analyzing instructions in units of traces.

In a trace processor, the backend as well as the frontend are built to operate in units of traces. A distributed backend is constructed from many small, simple *processing elements* (similar to a multiscalar processor [14]). Each processing element resembles the execution engine of a modest superscalar processor. Using small, simple execution engines enables them to be built to operate at higher clock rates. Replication is used to provide high aggregate throughput.

Traces are dispatched, one per processing element, to the distributed backend. Each processing element executes instructions from its own trace. The traces are part of one single

program, so there are data dependencies among traces. Values generated by traces are communicated between processing elements and all original program semantics are obeyed. The communication of values between processing elements has a greater latency than the communication of values within a processing element. This extra latency is the penalty for partitioning the execution resources.

1.3 Thesis Contributions

Incorporating a trace cache into a processor's frontend raises a number of implementation issues and opens up new opportunities for increasing performance. One important issue is that the dynamic nature of traces causes the trace cache to be sensitive to the processor's workload. The trace cache must learn dynamic behavior, which limits its benefit when executing new regions of code. The dynamic nature of traces also means the working set of traces is much larger than the comparable static representation, leading to capacity miss problems. Another important issue is that a new approach to branch prediction must be used in order to provide a bandwidth sufficient to take advantage of the fetch bandwidth of the trace cache. The trace cache can also enable a whole range of optimizations that dynamically transform programs to optimize specific dynamic sequences of instructions for implementation-specific hardware.

The proposed mechanisms are built to take advantage of the specific behavior of the trace cache and the fundamental paradigm of having a processor function in the unit of traces of instructions. This thesis consists of three main parts, each dedicated to a different mechanism to incorporate with the trace cache. Together these mechanisms enable the full

potential of the trace cache to be realized, and produce a high-performance, robust frontend for trace processors.

In the first part, I propose trace pre-construction, to augment trace caches by performing a function analogous to prefetching. The trace pre-construction mechanism observes the processor's instruction dispatch stream to detect opportunities for jumping ahead of the processor. After doing so, the pre-construction mechanism fetches static instructions from the predicted future region of the program, and constructs a set of traces in advance of when they are needed. Trace pre-construction can significantly increase both the performance of the trace cache and the robustness of the trace cache to varying workloads. Trace pre-construction requires sophisticated hardware, but does not affect the complexity of the main processor core.

In the second part, I propose a new class of hardware optimizations that transform the instructions within traces to increase the performance of the processor's execution engine. Traces are "pre-processed" for both optimizing common dynamic instruction sequences and to utilize implementation-specific execution resources. I propose three specific optimizations: instruction scheduling, constant propagation, and instruction collapsing. Together these optimizations enable the processor to exploit significantly more instruction-level parallelism.

In the third part, I propose a new type of control predictor to match branch prediction throughput with the bandwidth of the trace cache. I propose a next-trace predictor that treats the traces as basic units and explicitly predicts sequences of traces. The predictor collects

histories of trace sequences (paths) and makes predictions based on these histories. The basic predictor is enhanced to a hybrid configuration that reduces performance losses due to cold starts and aliasing in the prediction table. The Return History Stack is introduced to increase predictor performance by saving path history information across procedure call/returns.

The pre-construction and pre-processing mechanisms take advantage of an extended pipeline organization enabled by the trace cache. An autonomous engine analyzes the program independently of the main processor. This engine performs work on the program that is encoded, along with instructions, into traces that are placed into the trace cache. The trace cache decouples this new engine from the main processor core. The main processor core can utilize work performed by this new engine to increase the efficiency of instruction fetch and execution. A full description of this pipeline organization is given in the next section.

I believe that the trace processor organization is a very promising microarchitecture, and the work of this thesis complements it well, but this work can be viewed as applicable in a larger context. The work in this thesis is directly applicable to any processor organization that uses a trace cache. Much of this work is also indirectly applicable to any processor organization that uses larger blocks of work in the frontend.

1.4 An Extended Pipeline Model

Initial work in developing sophisticated new frontend mechanisms made it clear that it was important to rethink how we logically view the processor pipeline. This led to a new

extended pipeline organization and a framework for a range of mechanisms for optimizing processor performance.

The extended pipeline model contains a new pipeline, distinct from the fetch and execute pipelines (see Figure 1-4). I refer to the pipeline as a pre-processing engine, as it works on instructions before they are fed into the normal processing phases. The trace cache provides an opportunity to decouple this pre-processing engine from the traditional processor core. The pre-processing engine analyzes the program and encodes information into traces that it places in the trace cache. The main processor core fetches instructions from the trace cache and can take advantage of the extra information that is encoded along with the instructions in the trace. This pipeline organization is similar to the organization used by the CRISP processor [11]. The pre-processing engine is analogous to the prefetch and decode unit in the CRISP processor, and the trace cache performs the same role as the decoded instruction cache does in the CRISP processor.

The new pipeline organization takes advantage of two characteristics of traces. First, a valid trace can be placed into the trace cache at any time, independently of what the rest of the processor is doing. Second, the instructions within a trace need not be identical to the instructions specified in the static program representation, just functionally equivalent.

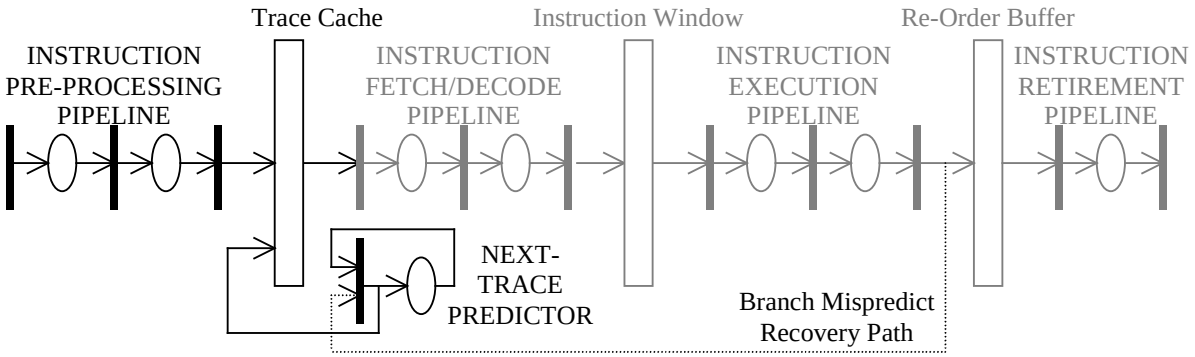


Figure 1-4 New pipeline organization with "pre-processing."

The pre-processing engine can be used for implementing many different functions. Using traces as the intermediate representation enables optimizations that encompass many instructions. The concept of the pre-processing engine emerged while I was developing the mechanism to pre-process traces (hence the overlap of nomenclature). As I started exploring trace pre-construction I realized that the pre-processing engine approach could be used for a whole range of mechanisms. Beside pre-construction and pre-processing, the new pipeline organization could be used for other optimizations like data memory prefetching.

This new organization enables sophisticated mechanisms to be incorporated into the processor with minimal impact on the tightly tuned processor core. This is important, as performance is very sensitive to the length of the pipelines in the processor core and the clock speed of the processor core. The pre-processing engine does not impact the latency to resolve mispredicted branches or the latency to redirect fetch and refill the main processor pipeline.

1.5 Related Previous Work

In this section I discuss the previous work that relates to each of the three main mechanisms I propose in this thesis.

1.5.1 Previous work related to trace pre-construction

Trace pre-construction builds on a large body of work related to pre-fetching. Over the past few decades the speed of processors has grown significantly faster than the speed of large memory structures. Hierarchical memory structures and prefetching have been extensively utilized to address the discrepancy between processor and memory speeds. In the area of prefetching, more work has been focused on prefetching of data than instructions. The temporal and spatial locality of instructions has allowed hierarchical memories to perform very well, reducing the need for sophisticated prefetching. There is still a sizeable body of work on instruction prefetching using both software and hardware based approaches.

A number of software approaches to reduce the need for instruction prefetching and/or enable simple instruction prefetching approaches to perform well have been proposed [42][30][61]. Compiler directed prefetching of instructions has also been proposed in a number of works [66][27].

There have been many proposed hardware schemes to support instruction prefetching based on spatial locality. A simple approach that is commonly applied is using long cache lines to give some degree of implicit prefetching [44]. Prefetching sequential cache lines have been studied in a number of works [53][21][55].

Other hardware instruction prefetching mechanisms have included prefetching the targets of branches. Smith and Hsu [55], and Pierce and Mudge [43] propose fetching the target of conditional branches that are not taken. The CRISP processor [11] incorporated an autonomous prefetch mechanism that could sequence ahead of the processor and prefetch

across calls and branches. Attempting to predict taken branches relatively far ahead in the instruction stream and to prefetch from the target was proposed by Young and Shekita [69]. Reinman, Austing and Calder [47] propose letting the branch predictor sequence ahead of the processor which could be used to provide a small degree of instruction prefetching.

The pre-construction engine I propose attempts to sequence significantly far ahead of the processor. To do this the pre-construction engine makes use of a policy for jumping ahead of the processor first proposed for the Dynamic Multithreading architecture [2] by Akkary and Driscoll.

1.5.2 Previous work related to pre-processing

I consider three pre-processing optimizations, instruction scheduling, constant propagation and instruction collapsing. These optimizations all borrow from established work in compilers as well as previous work in dynamic optimizations. I originally presented a description of the pre-processing optimizations at the 5th International Symposium on High-Performance Computer Architecture [24]. Very similar work was done independently around the same time by Friendly, Patel and Patt [17].

There has been extensive research into instruction scheduling within compilers. Instruction scheduling has been incorporated to optimize for specific pipeline organizations and to accommodate long latency operations [20][24][1]. For both superscalar and VLIW architectures it has been proposed that the compiler can schedule code to take advantage of parallel hardware to increase ILP [7][6][50].

The scheduling optimizations I consider are similar to the work in dynamically grouping instructions proposed in a number of works. Melvin et al. [32] proposed dynamically grouping independent instruction together to make use of parallel issue hardware. Franklin and Smotherman [15] proposed dynamically constructing VLIW instructions (stored in a shadow cache). Further work in the area includes the work with DIF caches [37].

Constant propagation/constant folding is a common optimization that is utilized in every optimizing compiler [1]. It is an intuitive optimization that can be easily implemented in the compiler without causing any side effects.

Data dependence collapsing has been proposed as a way to increase parallelism by collapsing data dependent operations into a single more complicated operation. Data collapsing first appeared in the context of floating-point arithmetic in operations like the fused multiply-add operation that is present in many instruction sets [34]. S. Vassiliadis et al. [63] proposed a mechanism to collapse dependent pairs of fixed-point arithmetic or logical operations. This work was continued by Phillips and Vassiliadis [45]. Performance studies to evaluate the potential of data collapsing of consecutive instructions were carried out in a number of works [28][62]. Y. Sazeides et al. [50] studied more relaxed forms of data collapsing. They found that 29% to 47% of instructions could be collapsed leading to a substantial increase in instruction-level parallelism. They also found that a majority of collapsing involved non-consecutive instructions.

1.5.3 Previous work related to next-trace prediction

Next-trace prediction builds on previous work in branch prediction. It incorporates aspects of advanced branch predictors, including work in path-based histories. It also incorporates aspects of previously proposed methods to predict multiple branches in a single cycle, multiple-branch prediction.

Branch prediction in some form is a fundamental part of next-trace prediction (either implicitly or explicitly). Hardware branch predictors predict the outcome of branches based on previous branch behavior. At the heart of most branch predictors is a Pattern History Table (PHT), typically containing two-bit saturating counters [54]. The simplest way to associate a counter with a branch instruction is to use some bits from the address of the branch, typically the least significant bits, to index into the PHT [54]. If the counter's value is above some threshold the branch is predicted taken. When the branch is later executed, the counter is incremented or decremented depending if the branch was taken or not.

Correlated predictors can increase the accuracy of branch prediction because the outcome of a branch tends to be correlated with the outcome of previous branches [39][68]. The correlated predictor uses a Branch History Register (BHR). The BHR is a shift register that is usually updated by shifting in the outcome of branch instructions -- a one for taken and a zero for not taken. In a global correlated predictor there is a single BHR that is updated by all branches. The BHR is combined with some bits (possibly zero) from a branch's address, either by concatenating or using an exclusive-or function, to form an index into the PHT. With a correlated predictor a PHT entry is associated not only with a branch instruction, but

with a branch instruction in the context of a specific BHR value. When the BHR alone is used to index into the PHT, the predictor is a GAg predictor [68]. When an exclusive-or function is used to combine bits from the BHR and the branch's address, the predictor is a GSHARE predictor [31]. GSHARE has been shown to offer consistently good prediction accuracy.

Nair [36] proposed "path-based" prediction, a form of correlated branch prediction that has a single branch history register and prediction history table. The innovation is that the information stored in the branch history register is not the outcome of previous branches, but their truncated addresses. To make a prediction, a few bits from each address in the history register as well as a few bits from the current branch's address are concatenated to form an index into the PHT. Hence, a branch is predicted using knowledge of the sequence, or path, of instructions that led up to it. This gives the predictor more specific information about prior control flow than the taken/not taken history of branch outcomes. Bennett, Sharma, Smith and I [21] refined the path-based scheme and applied it to next task prediction for multiscalar processors (building on previous work in next task prediction by Franklin [14] and Breach [3]). It is an adaptation of this multiscalar predictor that forms the core of the path-based next-trace predictor presented here.

In order to support simultaneous fetching of multiple basic blocks, multiple branches must be predicted in a single cycle. A number of modifications to the correlated predictors discussed above have been proposed to support predicting multiple branches at once. Pnevmatikatos, Franklin, and Sohi proposed a Multiblock prediction mechanism that implicitly predicts multiple branches with a single prediction [46]. The approach in

Multiblock prediction of working in units that encompass multiple branches is similar to the approach taken in implementing the next-trace predictor proposed in this thesis. A specific form of multiblock prediction is next task prediction [14][58][22][3] in multiscalar processors. Franklin and Dutta [12] proposed subgraph oriented branch prediction mechanisms that uses local history to form a prediction that encodes multiple branches. Yeh, et al. [67] proposed modifications to a GAg predictor to multiport the predictor and produce multiple branch predictions per cycle. Rotenberg et al. [48] also used the modified GAg for their initial trace cache study.

Recently, Patel et al. [40] proposed a multiple branch predictor tailored to work with a trace cache. The predictor attempts to achieve the advantages of a GSHARE predictor while providing multiple predictions. The predictor uses a BHR and the address of the first instruction of a trace, exclusive-ored together, to index into the PHT. The entries of the PHT have been modified to contain multiple two-bit saturating counters to allow simultaneous prediction of multiple branches. The predictor offers superior accuracy compared with the multiported GAg predictor, but does not quite achieve the overall accuracy of a single branch GSHARE predictor.

Chapter 2
Basic Trace Processor

In this chapter I present the organization of the basic trace processor into which I incorporate novel frontend mechanisms. I first present a general overview of the trace processor organization. Then I present the specific configurations I use for this thesis. Finally, I discuss the simulation methodology and benchmarks used to generate the results presented in this thesis.

2.1 Trace Processors

Trace processors are based on hardware mechanisms that allow the processor to predict and fetch a long sequence of instructions, a *trace*, in a single cycle. Traces are dispatched, one per processing element, to a distributed backend. Figure 2-1 shows a top-level diagram of a trace processor. The principles and main components of a trace processor will be discussed in the following sections.

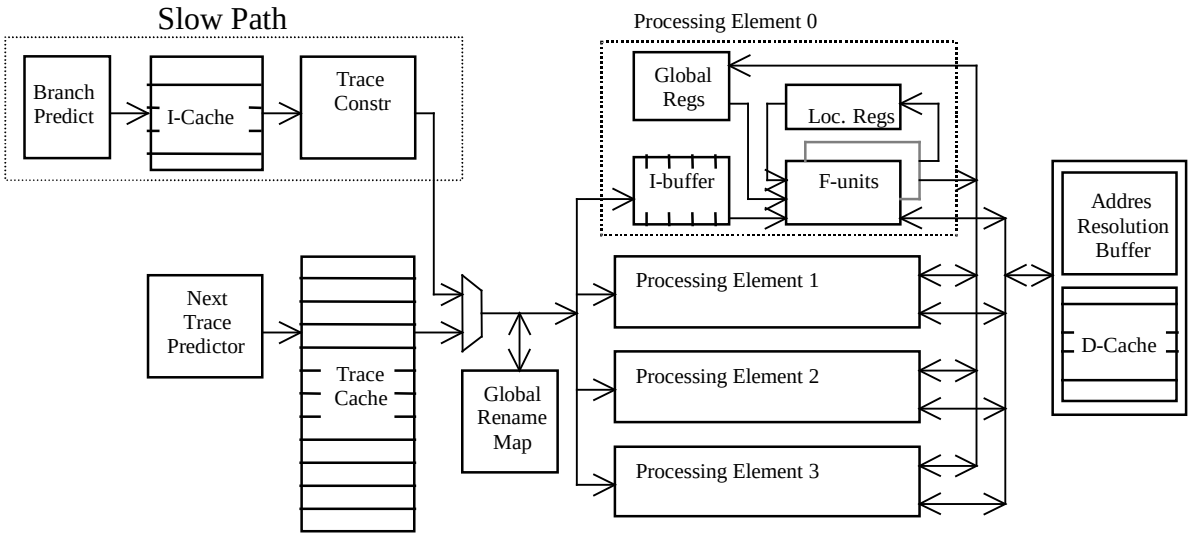


Figure 2-1 Trace processor hardware.

2.1.1 High level sequencing and wide fetch

Associated with the trace cache is a trace fetch unit, which fetches a trace from the cache each cycle. To do this in a timely fashion, it is necessary to predict what the next trace will be. A straightforward method is to predict simultaneously the multiple branches within a trace. Then, armed with the last PC of the preceding trace and the multiple predictions, the fetch unit can access the next trace. A more efficient approach to prediction is to treat the traces as the basic units of prediction and explicitly predict sequences of traces. This can be implemented with the next-trace predictor proposed Chapter 5.

A single entry in the trace cache holds an entire trace of instructions. The trace cache is indexed using the prediction information returned by the trace predictor. Thus, an entire trace consisting of multiple basic blocks is fetched in one clock cycle. This gives a very high bandwidth fetch path [49].

When the next trace of a program is not in the cache, the “slow path” hardware (Figure 2-1) constructs a trace, using branch prediction as needed to predict an entire trace. The trace construction performs some necessary transformations of the trace so that the specialized processing elements can execute the trace. This includes performing pre-renaming of registers contained in traces (described in the following section). This newly constructed trace is dispatched to an idle processing element to be executed. The total penalty for constructing a trace with the slow path hardware is variable. There is at least one cycle latency per basic block (more cycles if the instruction cache misses) to fetch the needed instructions. One additional cycle of latency past the fetching of the last basic block is modeled to account for trace construction transformations.

2.1.2 Register rename logic

Each trace has a *live-in* and *live-out* set of registers [65][49]. The *live-in* set of registers are those architected registers that are read before being written in a trace. The *live-out* set of registers are those architected registers for which a trace generates new values. The live-in and live-out registers in trace processors are equivalent to the live-in and live-out registers in multiscalar processors [14][58][3].

Within a trace some register values remain entirely *local*, generated and consumed in the trace and never used by later instructions. A register value is known to be local when a trace writes to a given architected register, uses that register, and then overwrites a new value to the same architected register.

Pre-renaming of local register values can be performed in the trace construction hardware off the critical path. Pre-renaming consists of mapping the architected registers of the instructions within a trace to a new trace-level name space. During the pre-rename, dependences within a trace are encoded and the set of live-in and live-out registers are determined. At time of dispatch the rename logic needs to map live-in and live-out registers to global names [65].

2.1.3 Distributed execution engine

The execution resources of a trace processor are divided among its processing elements (Figure 2-1). The distributed execution engine design builds on the work of multiscalar processors [14][58][3]. Each processing element has its own register file, instruction window and execution resources. Zero-cycle bypassing between execution units occurs only within individual processing elements, reducing the complexity and latency of the bypass logic. The goal of distributing the execution resources is to develop smaller, simpler execution engines that run with faster clocks than large centralized designs [58].

The register file in trace processors is hierarchical. Each processing element has a local register file to which every instruction in the trace writes its result. Each processing element also has a copy of a replicated global register file. Instructions with live-in source operands read from the local copy of the global register file. Instructions with live-out destination operands write to the local register file, the local copy of the global register file, and broadcast to all other global register files. There is a longer latency for global register

communication, but the ability to build tightly coupled processing elements with high clock speeds combined with the natural locality of data communication compensate for this latency.

Memory dependencies are handled in the same manner as they would be in a superscalar processor. I consider an aggressive approach where loads are allowed to issue before all previous stores are resolved. This reduces unnecessary stalling of loads by speculating that all loads are independent. Hardware must detect dependence violations caused by loads going early. The processor has selective reissue logic to reissue loads and all dependent operations when a load is determined to have violated true dependences. The dependence checking can be based on central Load/Store Queue or an Address Resolution Buffer (ARB) [16].

2.2 Trace Caches

2.2.1 Trace selection

Traces are formed by recording dynamic sequences of instructions observed in the processor. While observing the instruction stream, the trace selection heuristic will determine at some point that a trace is complete. That dynamic sequence will then be recorded as a trace and the subsequent instructions will become the beginning of a new trace. The heuristics used to partition the instruction stream into traces has a significant impact on all elements of the trace processor's front-end. They affect the probability that the processor will be able to reuse a trace and the number of traces competing for space in the trace cache.

The instruction supply bandwidth of a trace cache is a function of the hit rate and the number of instructions supplied in the case of a hit. There exists a tradeoff in trace selection between the desire for long traces to increase the benefit of the trace cache in the case of a hit, and the desire for short traces to increase the hit rate of the trace cache. Using longer traces increases the number of instructions that can potentially be supplied by the trace cache per cycle. On the other hand, using shorter traces can increase the probability that the trace cache will contain useful traces. For example, the probability that a specific 128 instruction long sequence will be repeated in a given period of time is lower than the probability that one or more of the corresponding 8 different 16 instruction long sequences will be repeated. But, if the 128 instruction long sequence is repeated, delivering all the instructions in a single cycle provides higher bandwidth than requiring 8 cycles.

The trace selection heuristic will likely produce traces of varying length in order to capture common instruction sequences. We only consider trace cache implementations where each entry of the cache holds a single trace and has fixed resources adequate to hold a trace of some maximum length. In such an implementation, traces of length less than the maximum length will waste potential resources. It is therefore important to consider both the maximum and average trace lengths. The size of the trace cache is a function of the former while the effective fetch bandwidth of the trace cache is a function of the latter.

Besides determining the appropriate size of traces, the trace selection heuristic must carefully determine where traces start and stop. Traces in the trace cache are only useful if they perfectly match the instructions immediately needed by the processor. There are two requirements of a match. First, the trace must start with the next instruction needed by the

processor. Second, the path represented in the trace must correspond to the expected path the processor will execute.

2.2.1.1 Trace alignment problem

When a dynamic sequence of instructions is observed repetitively there is an opportunity to take advantage of the trace cache. But, just because a sequence is observed repetitively does not mean that the trace cache will be able to provide the needed instructions. This occurs because of the *trace alignment problem*.

The trace processor breaks a dynamic sequence of code into traces, according to the trace selection heuristic. For any given sequence of code there are many valid ways to divide it into traces, depending on where the first trace starts (see Figure 2-2). This trace alignment problem reduces the opportunities to use traces in the cache and also increases the number of traces competing for entries in the cache.

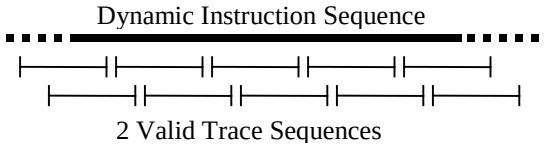


Figure 2-2 Illustration of different possible trace alignments.

An example of the trace alignment problem is a loop with 15 instructions in the body of the loop and a maximum trace length of 16. Assume a trace selection policy of packing 16 instructions into every trace is used and the first trace starts with the first instruction of the first iteration of the loop. The first trace observed will contain the instructions from the first iteration of the loop and one instruction from the second iteration of the loop. The next trace will contain the last 14 instructions of the second iteration and two instructions from the third

iteration and so on. The same dynamic instruction sequence is seen repetitively, but the trace cache can not provide the needed instruction until the 17th iteration of the loop where the needed instructions will match the first trace. This code sequence will also produce 15 different traces.

2.2.1.2 Trace selection heuristics

The trace selection heuristic determines the traces that should be constructed from the dynamic instruction sequence by using some set of rules to determine when a trace should terminate. One rule that must be used is the maximum length restriction; when a trace reaches a maximum length it is terminated. Trace selection rules can include stopping at specific types of instructions (i.e., return instruction). Trace selection rules can also include more sophisticated algorithms that include state to record instructions observed in the past. In developing the rules for trace selection it is important to consider the complexity of implementing them in hardware.

Trace caches work because the same sequence of code is executed multiple times. This occurs due to two common programming constructs: loops and common subroutines. In developing trace selection heuristics, it is important to consider these underlying constructs. Observing the backward branch that forms a loop most easily identifies loops. Both the calls to, and the returns from subroutines, are easily observable in the instruction stream.

There are a number of ways selection heuristics can help address the trace alignment problem. One way to improve alignment is to have some arbitrary *stopping points* (where traces will always terminate) in the static program. These points help reduce the trace

alignment problem by forcing different sequences to line up following the stopping point. These stopping points need not be added to the program, but instead achieved by taking advantage of constructs already in the program. For example, the trace selection heuristic can terminate traces at a specific instruction type such as indirect jumps. Another way to improve alignment is to have points in the static program where the number of possible alignments is limited. This can be achieved with having points in the program where traces are forced to terminate a multiple of *N* instructions past the point. This reduces the number of possible alignments past the point to the maximum trace length divided by *N*. This approach does not reduce the alignment problem as much as the stopping point approach, but it does not reduce the average trace length as much either.

In this thesis trace selection is based on four simple rules:

- **Maximum Trace Length**. I consider two different maximum trace lengths, 16 and 32 instructions. A maximum length less than 16 probably does not make sense, as the trace cache would not be able to provide significantly higher bandwidth than a traditional instruction cache. I do not consider longer traces than 32 instructions even though interesting design points may exist. Working at such coarse granularities raises a range of issues, such as wasted space in the trace cache and trace granularity retirement, that must be addressed before long traces will be practical.

- **Stop at Indirect Jumps**. Traces are terminated by any indirect jump instruction. In the instruction set the processor executes, indirect jumps are encoded as instructions

that jump to an address contained in a general-purpose register. The most common form of indirect jump is a return from subroutine instruction. The main reason for stopping at indirect jumps is to make unique identification of traces simpler. By forcing traces to end at indirect jumps, traces can be uniquely identified by their starting address and one bit to encode taken/not taken for each embedded branch. This can lead to efficient encoding of trace names. Another reason for stopping at indirect jumps is to introduce stopping points in a program to force good trace alignment.

- **Loop/Call Alignment**. An important trace selection threshold is to limit the number of initial trace alignments for loop iterations and subroutines. This is implemented with a simple heuristic that makes traces stop a multiple of *N* (where *N* is 2,3,4,..) instructions past a subroutine call instruction, backward branch instruction or backward jump instruction. Simply stopping at calls and backward branches (making *N* equal to the maximum trace length) would limit the number of initial trace alignments for subroutines and loop iterations to a single point, but also reduces the average length significantly. A compromise is to set *N* to one fourth of the maximum trace length. This limits the number of initial start points while reducing any individual trace by at most one fourth of the maximum trace length (and reducing the average by much less). Figure 2-3, discussed in the next section, shows the tradeoff of varying values of *N*.

- **Loop Threshold**. For large loop bodies the penalty (in terms of reduction in average trace length) is minimal for simply stopping at the backward branch that forms the

loop. For these loops, it makes sense to end at the backward branch and have each iteration start with the same trace alignment. Determining if the penalty is small requires determining the number of dynamic traces that form the loop body; if it is large then the impact of shortening the last trace is minimal. Precisely determining this is very hard, but a simple approximation can be used. The approximation used is to look at the static distance of the backward branch (static size of loop body) and determine if it is greater than some threshold. If the static distance is greater than the threshold, the loop is terminated. A threshold of around 96 static instructions performs well (see Figure 2-3 from the next section).

2.2.1.3 Impact of trace selection heuristics

To understand the tradeoffs in trace selection, a range of heuristics were studied. For these studies I look at two of the SPECint95 benchmarks, *gcc* and *go*. These two benchmarks have the largest executable images and contain an order of magnitude more unique traces than any of the other benchmarks. For each of these benchmarks, I present the average trace length and the number of unique static traces (see Figure 2-3). The number of unique static traces is a good indicator of the amount of pressure on the trace cache and other dynamic mechanisms in the processor.

The points for the simplest trace selection heuristics -- pack all traces with the maximum number of instructions -- are off the graphs in Figure 2-3 (their values are given at the top of the graph with arrows). By incorporating the arbitrary stopping points of indirect jumps (heuristic [16,32],1,0) the number of unique traces is significantly reduced at the cost of some

reduction in average trace length. Incorporating the alignment and threshold heuristics further reduce the number of unique traces while reducing the average trace length. The most restrictive heuristic has an order of magnitude fewer unique traces than the heuristic that packs the maximum number of instructions per trace.

Determining the optimal trace selection heuristic is impossible unless one considers a specific processor configuration. My work considers a range of different design configurations so I use a heuristic that I consider being a generally good compromise. I consider both maximum trace lengths of 16 and 32, with a loop call alignment of one fourth of the maximum length and a loop threshold of 96 (indicated in Figure 2-3 by the small arrows).

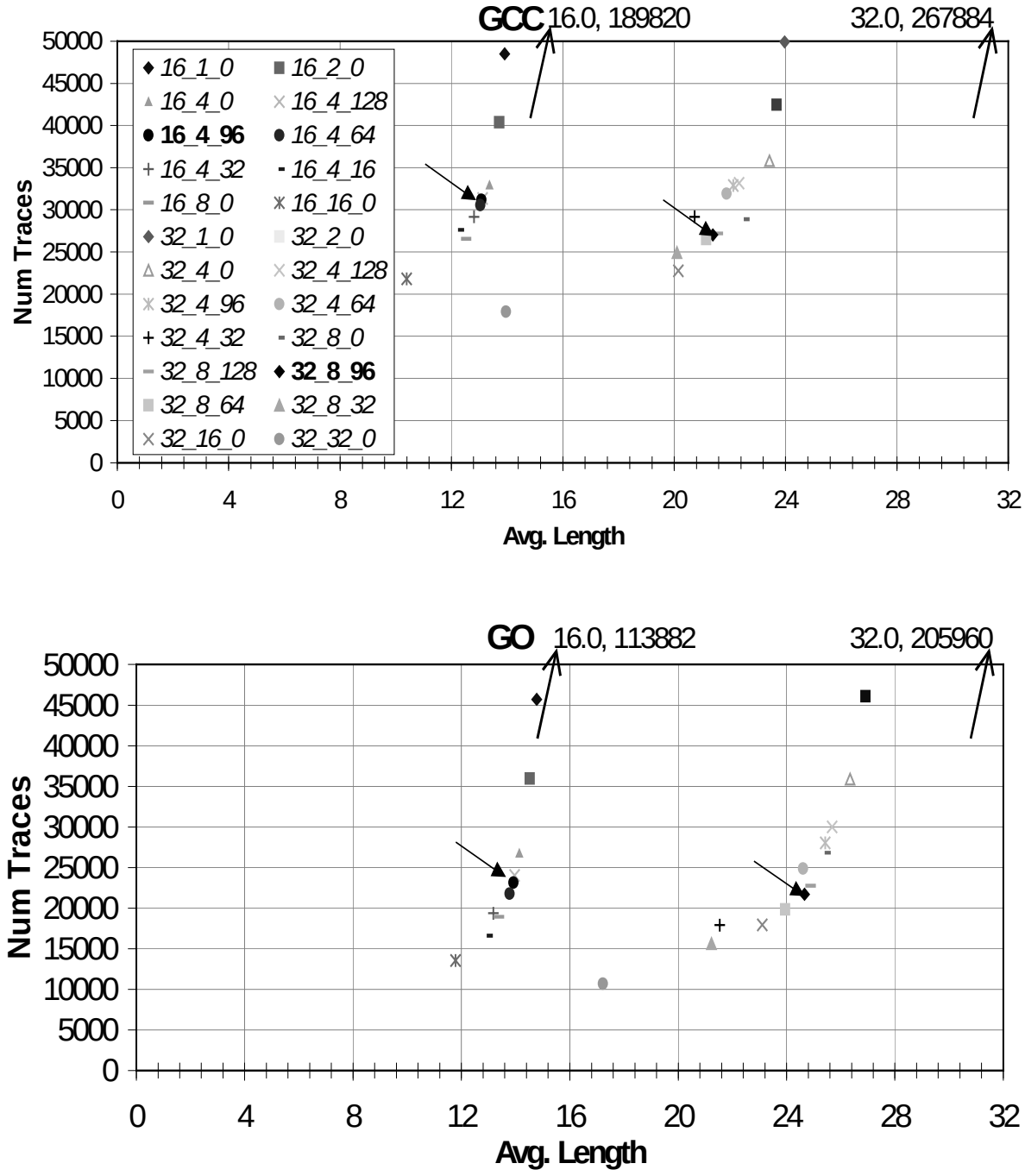


Figure 2-3 Trace selection tradeoffs presented as number of unique traces vs. average trace length. *Trace selection heuristics are encoded as <max length>_<alignment>_<loop_threshold>. A one (1) for alignment indicates the heuristic is not used. A zero (0) for threshold indicates that the heuristic is not used.*

2.2.2 Trace naming

To incorporate a trace cache into a processor requires some conventions for identifying traces. First, a unique method of identifying traces is required. The fewer bits used to identify a trace, the smaller the hardware structures that keep track of traces can be. Second, a method of indexing into the trace cache is necessary (a hashing function from the trace name to the potential position in the trace cache). This indexing method is important as it can significantly affect the performance of the trace cache by changing the probability of conflict misses.

2.2.2.1 Trace naming method

One theoretically possible way of identifying traces is to use the address of every instruction in the trace, but this name would be prohibitively long. A better way to uniquely identify traces is by the address of the first instruction of the trace and the outcomes of all embedded branches (1 bit per branch). An embedded branch is a conditional branch instruction that is not the last instruction of the trace. This naming system is possible because indirect jump instructions terminate traces; otherwise the targets of indirect jumps (a full word per jump) would be required to identify traces.

It is desirable that the trace names be a fixed width. To make the name a fixed width requires limiting the number of embedded branches that can be in a trace. This requires the trace selection heuristic to end traces at the first conditional branch after a maximum number of embedded branches has been observed. A simple way to achieve this is to set the

maximum number of embedded branches to one less than the maximum trace length (this means that the number embedded branches is never a factor in trace selection).

Limiting the number of embedded branches to a smaller reasonable value, for example 6 in the case of length 16 traces and 9 in the case of length 32 traces, would reduce the length of trace names and rarely affect trace selection. This is the approach that will be used for the remainder of this thesis. Even smaller values could be used, but I wanted to avoid having this limitation affect results.

2.2.2.2 Hashing trace names to index trace cache

A hashing function must be implemented to map any given trace to a single set of the trace cache. This is a very important function and can significantly affect the trace cache performance. The simplest approach is to use the low order bits of the address of the first instruction in the trace. There are two problems with this approach. First, multiple traces can originate from the same instruction (depending on which directions are taken for embedded branches), and often these traces will be accessed in close proximity. Therefore it is beneficial for different traces with a common first instruction to map to different sets in the trace cache. Second, only using low order bits may not be the best approach. The starting addresses of different traces will often vary by a multiple of the maximum trace length. This means that low order bits are not always a good way to differentiate traces.

The hashing function chosen for this thesis incorporates most of the bits from the starting address of the first instruction and the embedded branch outcomes by using the exclusive-or (XOR) function to combine multiple bits (see Figure 2-4). For a trace cache with *N* sets, the

index consists of $M = \log N$ bits. The hashing function consists of a number of steps. First, the low order M bits of the starting address (word address) are flipped so that the least significant bit of the address is in the left most bit position. The flipping causes the least significant bits – that are most likely to distinguish traces – to be combined with the bits that are least likely to distinguish traces. Second, this value is XORed with the branch outcomes, where the outcome (1 for taken, 0 for not taken) of the first branch is in the right most bit position and subsequent branch outcomes are in subsequent bits. A zero value is used for all bit positions past the last embedded branch. Third, this value is XORed with the next *M* bits of the address (least significant bit in the right most bit position). Last, this value is XORed with the next *M* bits of the address. In all 3*M* bits of the address of the first instruction are incorporated into the function.

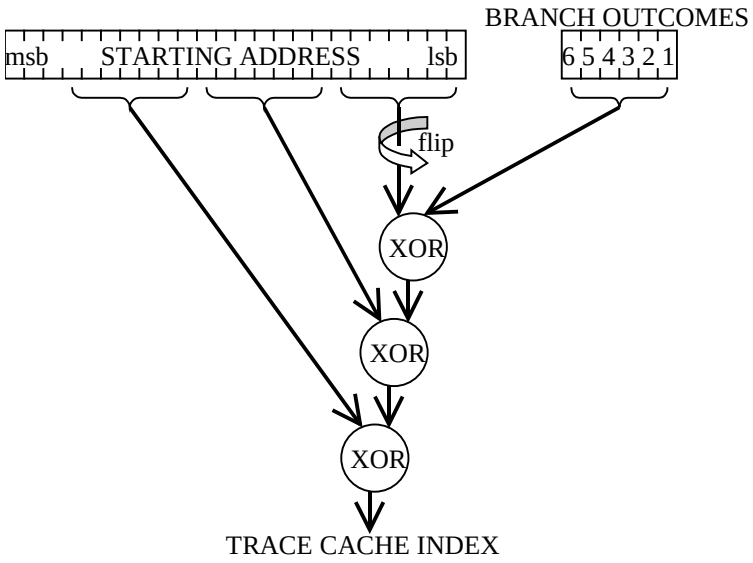


Figure 2-4 Function for forming trace cache index from trace identifier.

There is a significant amount of work that could be done in studying hashing functions for mapping trace identifiers into trace cache indices. The function presented here performs

well and is simple to implement, but better functions may exist. One can conceive of very sophisticated mapping functions that assign indices to intentionally avoid cache conflicts and which maintain mapping tables of indices to trace identifiers. Such an approach could potentially allow a direct mapped cache to perform as well as a fully associative cache. The expense would be the space and latency of the mapping tables, although the latency could most likely be avoided for the common read path (by having the trace cache index be the output of the next-trace predictor).

2.2.3 Trace cache management

2.2.3.1 Trace cache lookups

There are two ways that requests to the trace cache are generated. First, the trace cache is accessed when a processing element becomes available by retiring its current trace, and the next-trace predictor generates the access. Second, the trace cache is accessed when a processing element determines that it has an incorrect trace. In the second situation the processing element knows the outcome of some branches in the trace, including at least one branch whose behavior differs from the predicted trace. The processing element does not know what trace is correct, as it does not know what happens past the misprediction point of the current trace. When a processing element first detects that a trace is incorrect it refers to an “alternate trace” prediction from the next-trace predictor (this approach is unique to the trace processor design proposed in [49]). If the alternate trace matches for those branches that the processing element knows the outcome it is used, otherwise it is discarded. If the alternate trace matches, then a request for the specified trace is made to the trace cache. If the

alternate trace does not match, or in the case that the alternate trace itself is later determined to be incorrect, the good instructions of the trace and the known branch outcomes are passed to the slow path fetch engine so that a new trace can be constructed (see Figure 2-1).

When the trace cache is accessed, the hashing function is used to identify which line, or lines, of the cache may contain the requested trace. In a set-associative cache, where a trace may reside in one of multiple lines, the cache may be organized so that all the potential lines are accessed in parallel. Each line of the trace cache contains the instructions of the trace and a tag that identifies the trace. The tag is used to determine whether the requested trace is present. In the case of set-associative caches, the tag is used to identify which line, if any, has the requested trace.

If the requested trace is available in the trace cache, then the instructions of the trace are fed into the processor pipeline. The trace processor frontend is designed so that a sustained fetch rate of one trace per cycle can be maintained when needed. The large fetch bandwidth is useful for rapidly refilling the window after a control misprediction is detected.

2.2.3.2 Instruction supply from the “slow path” fetch engine

The mechanisms of a traditional instruction fetch engine (an instruction cache, branch target buffer and branch predictor) are present in the trace processor. There are two cases when this alternative fetch engine is used. The first case is when there is a miss in the trace cache. In this case the outcomes of all the branches in a trace are already predicted (and encoded into the trace identifier). The needed instructions are fetched from the instruction cache. The second case is when a misprediction is detected by a processing element. In this

case some good instructions and predictions are already available. The next block of instructions following the good instructions is fetched from the instruction cache. Branch predictions are used to predict any subsequent branches until the end of the trace is reached.

The slow path fetch engine feeds into a trace constructor. This mechanism determines when the end of a valid trace is reached and stops the fetch engine. This mechanism also performs some transformations on the instructions to make them a valid trace. The most important transformation is identifying and renaming register dependences within a trace. The other transformations may include adding a few bits of decode information per instruction and developing the trace identifier for the trace.

The predictors for the slow path fetch engine are very simple. PC indexed bimodal branch predictors and branch target buffers are used [54]. The next-trace predictor is a very aggressive predictor and this is where most of the prediction resources are dedicated. The slow path predictors are simple to reduce cost and complexity. Maintaining an accurate branch history register would be complicated given the transitions between trace granularity and instruction granularity fetching in the frontend. The branch predictor is also more likely to be used when a dynamic instruction sequence is seen for the first time, as the trace-level predictor will handle most regularly repeated sequences. In this situation a correlated predictor is not likely to perform well.

2.2.3.3 Trace cache updates

There are two logical policies for updating the trace cache. First, the trace cache can be updated whenever the slow path generates a trace. Second, the trace cache can be updated

when a trace generated by the slow path is determined to be good, which occurs at the retirement of the trace. Because the slow path is used for cases where the prediction has gone astray, the trace generated by the slow path may already be in the trace cache. It is therefore necessary in the case of a set-associative trace cache to check the cache prior to inserting a new trace to avoid having the same trace reside in different ways.

There are a number of tradeoffs between the two update policies. The update-at-retirement policy avoids placing potentially useless traces in the cache. But, some of the erroneous traces generated may be useful later when that path is required. The early update also reduces the latency of update, so if a trace is used again in a short period of time it may be available in cases where the cache would not have been updated by the update-at-retirement policy.

For this work I will only consider the update immediately policy. Preliminary studies showed this policy tends to perform better. More importantly, this approach works much better in conjunction with trace pre-construction than the update-at-retirement approach.

2.3 Base Machine Configuration

A trace processor design based on the one proposed by Rotenberg et al. [49] and described in Section 2.1 is used as the base processor design. Three different configurations are used throughout this thesis. The different configurations have most mechanisms in common and differ primarily on the size of the instruction window and the instruction issue

width. The configurations also differ on the maximum trace size they use; two use length 16 traces while one uses length 32 traces.

2.3.1 Instruction supply mechanisms

The default trace cache for all configurations is 2-way set-associative and has 512 lines (32 Kbytes of instructions for length 16 traces; 64 Kbytes of instructions for length 32 traces). For trace construction the processor uses a 16K entry branch predictor (indexed by low order bits of pc) and a 16K entry branch target buffer, these require 4 Kbytes and 64 Kbytes respectively. The instruction cache is 4-way set-associative with 128-byte line size and has a total capacity of 256Kbytes. The mechanisms in the slow path are sized relatively large to avoid having poor performance on the slow path make the trace cache look good.

The next-trace predictor is a hybrid predictor that combines a correlated predictor and a simple predictor. The correlated predictor has 64K entries and is indexed based on a hash function that incorporates the history of the last eight traces. The simple predictor has 32K entries and is indexed by only the history of the most recent trace. The predictor uses a Return History Stack. A detailed description of the predictor is given in Chapter 5. The predictor requires over 256 Kbytes of space for the straightforward implementation. This cost could be reduced by at least a factor of two with the reduced cost implementation discussed in Chapter 5. Further reducing the size of the predictor by a factor of two or four will have no effect for most of the benchmarks and only produce minor performance degradations for a couple of benchmarks (see results on sensitivity of predictor to size in Section 5.5.1)

2.3.2 Data memory subsystem

There is a single level-one data cache shared by all the processing elements in the processor. The level-one data cache has 4 ports of which any single processing element can only access 2 ports per cycle. The data cache is non-blocking and is write-back. The data cache has 64-byte lines, is 4-way set-associative and has a total size of 64Kbytes.

Memory operations are broken into two sub-operations at the processing element, address computation and the load/store operation. Address computation requires one cycle. The level-one data cache has a two cycle hit latency. This leads to an overall load-use latency of 2 cycles. A perfect level-two cache is modeled. The level-two cache has a 10-cycle hit latency. Contention for the level-two cache is not modeled.

Loads are allowed to issue before all previous stores are resolved. The processor has selective reissue logic to reissue loads and all dependent operations when a load is determined to have violated true dependences. A violation is detected when a store operation is performed. The cycle after a violation is detected the offending load is reset in the instruction window. It will then compete in following cycles for issue bandwidth. Instructions dependent on the load will reissue when they receive new values [49].

2.3.3 General execution engine configuration

Traces are dispatched, one per processing element, to a distributed backend. Each processing element is assumed to have enough functional units that any combination of operations, up to the issue width, can be issued in parallel in a given cycle. A real implementation would most likely use a more restrictive processing element where only

certain combination of instructions could be issued in parallel (based on observed utilization of functional units).

The latency of each operation is equivalent to the latency of the corresponding operation in the MIPS R10000 processor [33]. Each processing element has full bypasses internally and can support back-to-back dependent operations. For communication between processing elements there are global result busses. There are 8 total global result busses and an individual processing element can only use up to 4 in a single cycle. It takes a full cycle for global results to be broadcast on a result bus. If an instruction is executed in one processing element in cycle *K*, the result can be broadcast in cycle *K+1*, and dependent operations can be executed in other processing elements in cycle *K+2*.

2.3.4 Three processor configurations

The parameters that differ between the processor configurations are the length of traces, the number of processing elements and the issue bandwidth per processing element. Figure 2-5 gives an example of how performance (measured in Instruction Per Cycle) varies as a function of these three parameters for the benchmark *gcc*. Separate results are presented for length 16 and length 32 traces. The configurations are given as the aggregate window size and the aggregate issue bandwidth (grouped by window size). The number of processing elements is the aggregate window size divided by the trace length. The issue width per processing element is the aggregate issue width divided by the number of processing elements. The other benchmarks show similar sensitivities to the parameters.

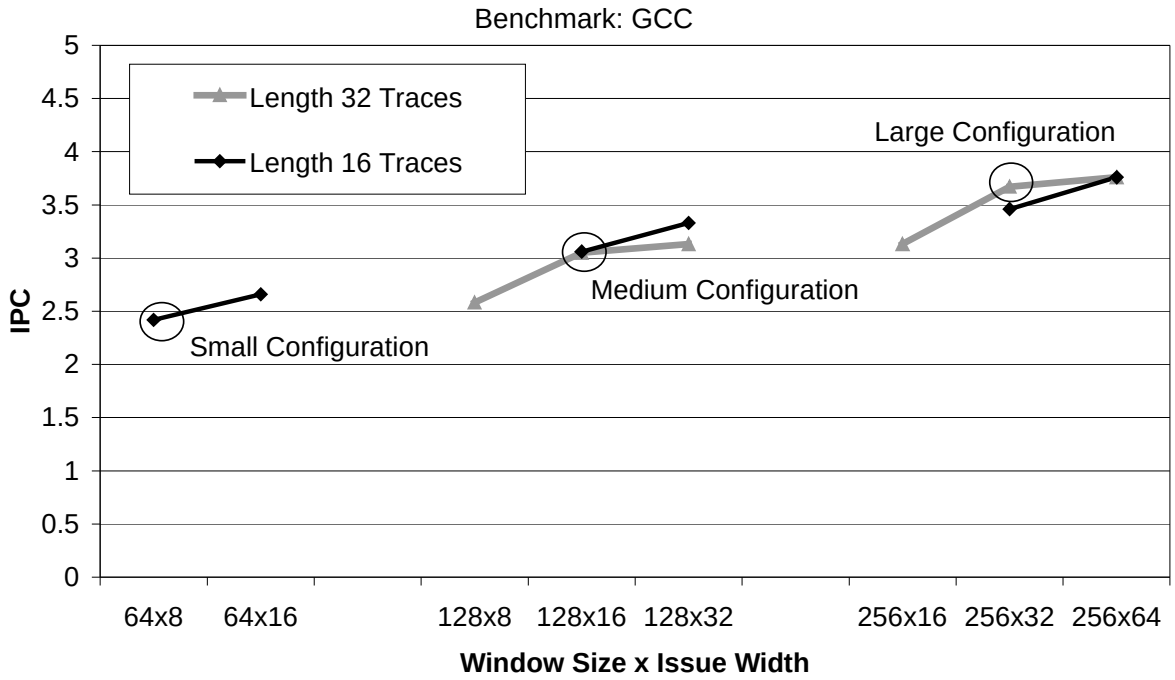


Figure 2-5 Performance as a function of execution engine configuration for *gcc*.

There are three processor configurations commonly used in the studies presented in this thesis. A *small* configuration, which uses length 16 traces and has 4 processing elements with 2-way issue per processing element (total window size of 64 instructions and total issue bandwidth of 8). A *medium* configuration, which uses length 16 traces and has 8 processing elements with 2-way issue per processing element (total window size of 128 instructions and total issue bandwidth of 16). And a *large* configuration, which uses length 32 traces and has 8 processing elements with 4-way issue per processing element (total window size of 256 and total issue bandwidth of 32).

2.4 Simulation Methodology

2.4.1 Simulator

This research requires extensive infrastructure. At the core is a very detailed execution driven simulator that models a trace processor. Eric Rotenberg and myself jointly developed the simulator. The simulator models all the major components of a trace processor and provides a cycle-level accurate timing model of execution.

The simulated processor executes the Simplescalar instruction set [5]. The Simplescalar instruction set is a vanilla RISC instruction set closely modeled on the MIPS instruction set. There is neither extra information in, nor special transformation done on, the programs to support the trace processor model.

The functional simulator available with the Simplescalar tool set is run in parallel with the timing simulator. The retirement stream of the timing simulator is compared against the functional simulator to guarantee correct functional behavior of the timing simulator. This approach proved extremely important in developing and debugging the simulator.

Verifying timing accuracy of the simulator is much more difficult. I believe the timing results of the simulator to be highly reliable, although I can not prove them to be absolutely correct. The simulator is cycle based, that is, there is a global cycle and each pipe stage (from the last back to the first) performs its work for each cycle and then the processor moves to the next cycle. Many asserts and checks are integrated into the simulator to detect timing anomalies (such as an instruction retiring too few cycles after being fetched). Running

simple hand coded micro-benchmarks and running standard benchmarks on processor configurations that resembled other “trusted” simulators provided sanity checks of the timing simulator. Having two separate individuals actively working on every part of the simulator also helped to provide checks.

The simulator runs on a variety of platforms including Sparc/Solaris, x86/Solaris and X86/Windows NT (with Interix). The majority of the results for this thesis were generated on PentiumPro (x86) systems running Windows NT with an Interix subsystem. Some results (those related to the benchmark m88ksim) were generated on Sun Sparc systems running Solaris.

2.4.2 Benchmarks

The SPEC95 integer benchmarks (SPECint95) are used for the performance evaluation studies in this thesis. The SPEC benchmarks are the most commonly used benchmarks for academic microarchitecture research. The integer benchmarks offer more challenges in achieving high instruction fetch bandwidth, as they contain smaller basic blocks. For this reason, the thesis focuses on the integer benchmarks. Key results for the SPEC95 floating-point benchmarks (SPECfp95) are provided in Appendix A. The SPEC benchmarks represent real programs but are smaller on average than many important commercial applications. The benchmarks are compiled with the Simplescalar compiler that is a derivative of *gcc*-2.6.3. The benchmarks are compiled with optimization level “O3.”

Unfortunately I am unable to run operating system code or observe the effects of context switches with the simulation infrastructure. This, combined with the relatively small

footprints of the SPEC benchmarks, limits the ability to fully stress the caches and dynamic prediction structures portions of the processor (all relying on dynamic learning). I use two approaches for dealing with this limitation. First, modeling smaller cache structures to increase the probability of conflicts, hopefully getting results similar to running larger workloads with larger caches. Second, periodically invalidating some portion of the cache lines to model context switches. This is important when studying ways of reducing the frequency and cost of trace cache misses.

For many of the SPECint95 applications reduced size inputs (derived from the traditional inputs) are used so that all or most of an application can be executed. I run each benchmark for at most 200 million instructions. This allows a reasonable portion of a program to be exercised, while keeping the simulation times reasonable. The specific inputs used and the number of executed instructions for each SPECint95 benchmark are given in Table 2-1.

Table 2-1 Input data sets for SPECint95 benchmarks.

Benchmark	**Input**	**Instructions executed**
compress	Self generated input based on parameters "400000 e 2231"	First 200 million instructions
gcc	genrecog.i	116,292,119 instructions
go	9 by 9 game	132,916,788 instructions
ijpeg	Input file Vigo.ppm, 4 encodings based on different encoding parameters and 1 decoding.	116,736,148 instructions
li	8 queens problem	First 200 million instructions
m88ksim	dcrand	120,478,991 instructions
perl	scrabbl.in	First 200 million instructions
vortex	*vortex*.raw	First 200 million instructions

Some of the key trace-level characteristic of the SPECint95 benchmarks are given in Table 2-2. The base performance on the SPECint95 benchmarks for the three processor configurations are presented in Table 2-3. Later results will often be presented as a percentage speedup over these base performance values.

Table 2-2 Characteristics of SPECint95 benchmarks.

Benchmark	Length 16 Traces		Length 32 Traces	
	Average Trace Length	**Number of Unique Traces**	**Average Trace Length**	**Number of Unique Traces**
compress	13.47	602	19.84	488
gcc	13.09	32529	21.41	28576
go	13.91	24382	24.58	23209
ijpeg	14.75	2833	26.75	2602
li	11.29	1138	18.20	924
m88ksim	12.44	3487	22.76	2984
perl	13.39	2574	23.12	2113
vortex	13.71	7579	23.20	5604

Table 2-3 Base performance on the SPECint95 benchmarks for the three processor configuration.

Benchmark	Performance (Instruction Per Cycle)		
	Small Configuration	Medium Configuration	Large Configuration
compress	1.68	1.85	1.94
gcc	2.42	3.06	3.67
go	2.05	2.50	2.94
ijpeg	2.75	3.34	3.95
li	2.50	3.21	4.14
m88ksim	2.59	3.08	4.26
perl	2.58	3.25	4.04
vortex	2.95	4.14	5.36

Chapter 3
Trace Pre-Construction

In this chapter I propose a new mechanism, trace pre-construction, to significantly increase the performance of the trace cache. Trace pre-construction augments the trace cache by performing a task analogous to prefetching. Pre-construction is implemented with a sophisticated mechanism that sequences ahead of the processor and generates likely future traces. The extended pipeline organization (as described in Section 1.4) is employed to decouple the pre-construction mechanism from the main processor core.

The dynamic aspect of traces, which enables the trace cache to provide high instruction fetch bandwidth, makes trace caches vulnerable to compulsory and capacity misses. The compulsory miss problem arises because traces are “learned” from observing dynamic program behavior; if a given dynamic trace has not been observed before, the trace cache will not be able to provide that trace. Each static instruction may occur in many different dynamic sequences, creating redundancy between and within traces. Because of this redundancy, the working set of traces is larger than the comparable static representation. Hence, there can potentially be a very large number of unique traces observed for a program, this exacerbates compulsory misses and causes capacity misses. Not only do the learning time and the large working set of trace caches limit their performance, they limit the robustness of trace caches to varying workloads and environments

A potential method for reducing the miss rate of trace caches is to "prefetch" into the trace cache. Prefetching has been shown to be effective at reducing the miss rates of instruction caches [53][55][69]. Prefetching is based on predicting an item to be requested from the cache in the future, and then fetching the item from lower-levels of the memory hierarchy. When prefetching is successful, the item is in the cache by the time it is needed, and a cache miss is avoided. The effectiveness of prefetching is dependent on the accuracy of predicting what will be needed and the timeliness of this prediction.

When applying the concept of prefetching to trace caches, the dynamic aspect of traces raises a number of issues. First, trace caches are not part of a true memory hierarchy, as there is no base level that contains all possible traces. Therefore the term "prefetching" is not entirely accurate, as there is nowhere from which to fetch complete traces. I use the term *pre-construction* of traces because potential traces need to be constructed from static instructions.

Second, predicting future traces is a difficult problem. Traces are identifiable by their starting instruction and the outcomes of branches within the trace. To be effective, the pre-construction mechanism must identify a region of the program the processor will reach and the dynamic paths that will be taken through that region. In addition, the pre-construction mechanism must also identify the trace *alignment* along each path. Two traces are aligned if one terminates where the next begins. For a single path through a region of code there are many possible sequences of traces that can be identified, depending on where the first trace starts (trace alignment is discussed in Section 2.2.1.1). If the trace starting points identified by the pre-construction mechanism do not match the starting points needed by the processor, the pre-construction effort will have been wasted.

Third and finally, there is the issue of timeliness. The pre-construction mechanism must stay sufficiently ahead of the processor to accommodate the high latency of constructing traces. The pre-construction mechanism must be responsive to the processor "catching up" to it; i.e., knowing when to give up on a region and move farther ahead of the processor. At the same time, the pre-construction mechanism must avoid getting too far ahead of the processor; the pre-construction mechanism should not tie up space in the trace cache with traces that will not be needed in the near future.

In this chapter I describe an implementation of a trace pre-construction mechanism. The mechanism attempts to sequence ahead of the processor and construct potentially useful traces from the static program representation. The hardware required to implement pre-construction resembles a small processor core of its own. Pre-construction fits into the new pipeline organization of the pre-processing pipeline. The hardware to perform pre-construction is decoupled from the main processor core. The trace cache acts as the mechanism that decouples the pre-construction hardware from the main processor core, while allowing useful work to be passed from the pre-construction hardware to the main core.

3.1 Incorporating Trace Pre-Construction

Trace pre-construction is incorporated into a trace processor frontend by adding two main components, a pre-construction engine and a pre-construction buffer (see Figure 3-1). The pre-construction engine fetches instructions from the instruction cache, constructs traces, and places them into the pre-construction buffer. The pre-construction buffer is accessed in parallel with the trace cache. Misses are avoided when the trace cache misses but the pre-

construction buffer contains the desired trace. In this case the trace is copied from the pre-construction buffer to the trace cache. The separate pre-construction buffer is used to avoid trace cache pollution (displacing traces the processor is currently using with traces the processor may use in the future).

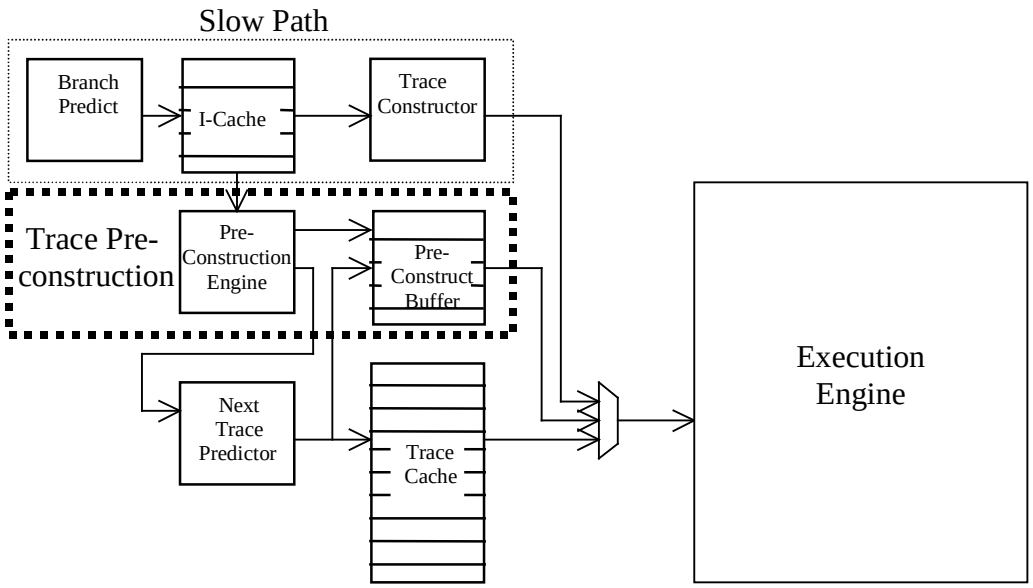


Figure 3-1 Trace processor with trace pre-construction.

The pre-construction engine could also "warm up" the next-trace predictor. When the pre-construction engine generates a new trace that does not already have an entry in the predictor, it could update the predictor with a likely candidate based on the pre-construction engine's heuristics. While integrating the pre-construction engine with the predictor appears to be a very interesting optimization, preliminary experiments I performed show minimal benefit from it (less than 0.1% impact on overall performance). Under certain circumstances I think the benefit would be more significant, such as cases when the predictor is suffering substantially from capacity misses. However, these do not occur in the benchmark set studied here. Therefore, I do not present results for using this optimization in this thesis.

3.2 Implementing Trace Pre-Construction

There are three major steps in trace pre-construction. The first is identifying *start points* for potential pre-construction *regions* (see Figure 3-2). A region is a portion of the program the processor is likely to reach in the future, and it is identified by the address of the first instruction, the start point. The pre-construction engine observes the instruction stream being dispatched to the processor core and predicts region start points. Once potential regions are identified, pre-construction begins for possible traces within the region. To solve the trace alignment problem (Figure 2-2), it is desirable that regions correspond to some natural boundaries of programs where the trace alignment is predictable.

The second step of trace pre-construction is fetching instructions from the regions. The pre-construction engine contains some number of small instruction windows, called *pre-construction windows*. The most recently identified regions are each assigned a pre-construction window. The pre-construction engine fetches instructions for each region, starting at the region start point, and fills them into the corresponding window. The instructions are buffered in a window because each instruction may be used many times in constructing different traces. The instructions in the window are linked together to form a control-flow graph (CFG). Typically multiple dynamic paths through each region will be fetched; precisely which instructions are fetched as part of a region and their order depends on the heuristics implemented.

The third step of trace pre-construction is identifying and constructing possible traces from each region. To do this the trace alignment must be predicted. Based on program

constructs and the trace selection heuristic used, the most likely starting points for traces are identified. Initially only a few trace starting points will be identified near the beginning of the region. Possible traces beginning at these initial starting points are constructed and placed in the pre-construction buffer. As traces are constructed within the region, subsequent trace starting points are identified.

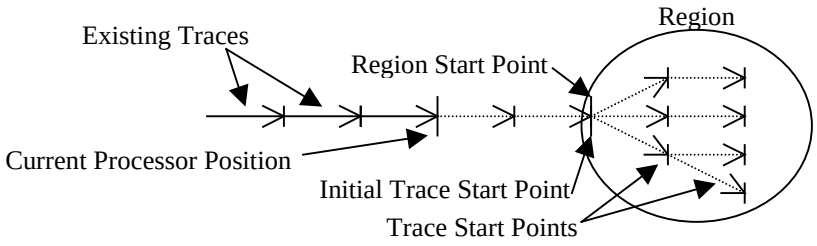


Figure 3-2 Overview of trace pre-construction (solid lines indicate traces thus far, dotted lines indicate future traces).

One important design decision in developing trace pre-construction is the degree to which it relies on learning dynamic behavior. By incorporating sophisticated learning mechanisms that dynamically learn the behavior of specific applications, trace pre-construction can be made more efficient. A possible learning mechanism might dynamically develop and maintain a graph representation of a program with priorities specified at every decision point. This information could be used to identify which region the processor is likely to reach in the near future and the paths that are likely to be taken within that region. The alternative is to implement trace pre-construction based on simple heuristics and implicit information encoded into the static program representation.

I focus on trace pre-construction using a minimal amount of dynamic learning. Implementations based on dynamic learning warrant more exploration, but they have a

number of drawbacks that led me to look at simpler alternatives. First, there are the obvious drawbacks of size and complexity of learning mechanisms. More importantly, with trace pre-construction based on dynamic learning there are issues of working set size and learning latency. These are the same issues that limit the effectiveness of the trace cache and lead to the need for pre-construction. Implementing a pre-construction mechanism that does not rely on dynamic learning increases the robustness of the processor; it enables the processor to gracefully handle large programs and frequent context switches.

3.2.1 Identifying and managing region start points

The pre-construction engine takes advantage of two common, easily identifiable constructs found in all programs: calls to subroutines and loops. When a call to a subroutine is dispatched to the processor, the return point is identified as a region start point (see Figure 3-3). When the call is executed there is a very high probability that the program will later reach the return point, easily identified as the static instruction following the call. When a taken backward branch is dispatched to the processor, the pre-construction engine assumes a loop and the exit point of the loop is identified as a region start point (see Figure 3-3). Although not all backward branches correspond to loops and not all loops exit to the instruction immediately following the backward branch, this heuristic works in most cases. These heuristics are similar to the heuristics proposed for identifying spawning opportunities in Dynamic Multithreading processors [2].

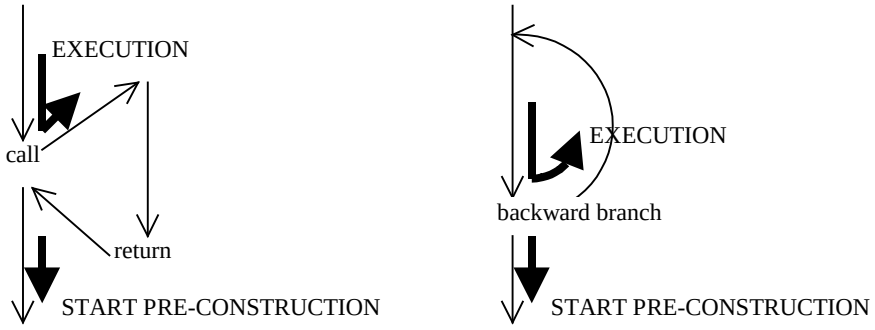


Figure 3-3 Identifying region start points.

The region start points corresponding to loops and subroutines exhibit an orderly nested behavior. The most recently observed backward branch or call corresponds to the next loop exit or subroutine return observed. This behavior makes it easier to manage and prioritize region start points. The more recent region start points should be given higher priority because they represent regions of code the processor will reach soonest. As the processor's execution passes through region start points, they can be discarded.

A small stack of region start points is maintained (see Figure 3-4). In order to stay ahead of the processor, it is necessary to observe the dispatch instruction stream, which includes speculative instructions. Start points are pushed onto the stack when a call or backward branch is observed on the processor's dispatch stream. When the stack fills up, the oldest entry on the stack will be discarded to make room for newer entries. To avoid redundancy, a new start point is not pushed if it corresponds to the same region as the current top of the stack. Start points are removed from the stack if they correspond to *misspeculation* or when the processor's execution has *reached* the region to which they correspond. Observation of the retirement stream of the processor determines when a start point should be removed.

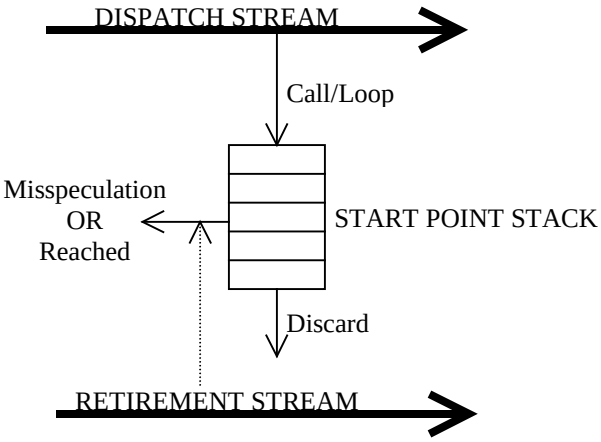


Figure 3-4 Start point stack.

Along with the start point stack there is an *active region buffer*. Each unique starting-point address in the start point stack has one entry in this buffer. The entries in this buffer have priorities corresponding to their highest priority entry in the start point stack. In addition the buffer also remembers the most recent regions for which all traces have been generated (such a region is considered *completed*), even after all the start point stack entries corresponding to them have been removed.

The pre-construction engine actively works on the highest priority, uncompleted, regions referenced in the active region buffer (these correspond to the highest priority start points). Remembering the regions for which pre-construction has completed reduces redundant work. The regions that have been recently completed are marked uncompleted if one of the traces they generated is replaced before being used. The method for supporting this is discussed when the pre-construction buffer is explained.

3.2.1.1 Why calls and loops?

Calls to subroutines and loop back edges are used to identify start points because they are common program constructs that are easily identifiable and whose behavior is well defined. They provide an opportunity to easily determine when to jump ahead, and where to jump, allowing the pre-construction engine to stay ahead of the processor. My work on thread selection for the Dynamic Multithreading (DMT) Architecture [2] suggested this approach. Pre-construction based on subroutines and loops performs well enough that I did not investigate alternative methods.

Not all subroutine calls and loop back edges are good start points. Occasionally the time from the call to the subroutine return or from the back edge to the loop exit is too long or too short. It is possible to augment pre-construction with mechanisms that dynamically learn which calls and back edges are good for jumping ahead of the processor. However, I experimented with such mechanisms and found that they offered minimal benefit.

Potentially, a compiler can identify better start points to direct pre-construction and encode this information into the program. The problem of identifying these points is similar to the problem of identifying tasks in a multiscalar compiler [64]. A compiler-supported version of pre-construction warrants more exploration, but I chose to focus on hardware-only implementations.

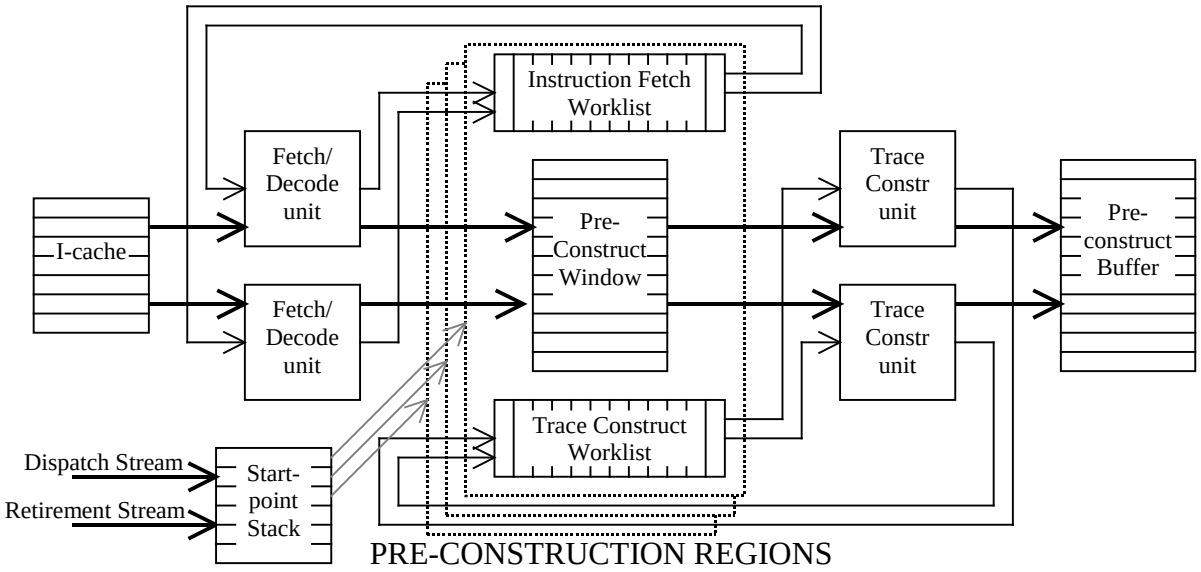


Figure 3-5 Pre-construction engine.

3.2.2 Fetching instructions contained in regions

Once the pre-construction regions are identified, the next step is to fetch the instructions from those regions. Each of these regions is assigned a *pre-construction window* to hold the fetched instructions. As they are fetched, the instructions in the pre-construction window are linked together to form a control-flow graph (CFG), with each static instruction only appearing once.

There can be one or more instruction cache ports available for pre-construction (I will focus on implementations that use only one). Corresponding to each instruction cache port is an instruction fetch and decode unit (see Figure 3-5). The regions share the instruction fetch and decode unit(s), with higher priority given to the region corresponding to the newest start point on the start point stack.

The first fetch from a region is for a cache line's worth of instructions beginning with the starting point instruction of the region. When a fetch is performed, the instructions are

placed into the appropriate pre-construction window, and one or more subsequently needed fetches are identified. Decoding the instructions and looking for conditional or non-conditional control-transfer instructions identifies additional fetch points. One cycle is required for the actual fetch (assuming instruction cache hit) and one cycle is required for the decoding of instructions, these two tasks are pipelined. The targets of control-transfer instructions are computed in the decode step and pushed onto a small worklist of needed fetches.

For each conditional branch, the bimodal branch predictor (table of 2-bit saturating counters indexed by branch address) in the slow path mechanism is referenced. The information from the branch predictor is used for two purposes. First, the pre-construction engine will only follow the dominant direction for highly biased branches. This reduces the number of paths that are followed. Backward branches that are highly biased in the taken direction most likely correspond to loops that will eventually exit, so both directions are followed in this case. Second, the favored direction of weakly-biased branches are remembered and encoded into the CFG. This information is used later to determine which traces to construct first. If there is no entry in the branch predictor, forward branches are assumed to be weakly-biased not taken and backward branches are assumed to be weakly-biased taken.

There is one worklist of instruction fetches per active region. The worklists use a simple algorithm to order fetches. The priority of the first fetch request is one (lower values have higher priority). The priority value is increased for every conditional branch observed to reach the current fetch point; it is increased by one if the branch corresponds to the favored

direction and by three if it corresponds to the non-favored direction. This priority scheme was determined to be a good balance of depth vs. breadth (the tradeoff is explored in Section 3.3.3.4). Instruction fetch requests with equal priority are handled in a first come first serve manner.

A design that does not use the branch predictor is also considered. Referencing the predictor requires extra read ports on the prediction table. Avoiding the need for these extra ports reduces the cost of implementing pre-construction. The results suggest that using simple static predictions degrades performance by a moderate, but possibly acceptable, amount (see Section 3.3.3.4). With minimal compiler support the performance would likely be comparable.

3.2.3 Decoupling instruction fetch from trace construction

The pre-construction windows are used to decouple the instruction fetch and trace construction mechanisms in the pre-construction engine. There are two obvious alternatives for this design. The first is to have no buffers; instructions would be directly fed to the trace constructors from the fetch mechanisms. This would reduce the size and possibly the latency of pre-construction. The problem of not having buffering is that some instructions may be used in many traces, and it is more efficient to buffer instructions than fetch them repetitively. The second alternative would be to use one large pre-construction window that is shared. This would be more flexible and possibly more efficient. But, separate pre-construction windows make management much simpler. Pre-construction windows are filled contiguously, and the only state needed for buffer management is a single pointer that

indicates where the next instruction should be inserted. When a region is completed or preempted, the entire buffer can be cleared by simply resetting the pointer to the head of the buffer.

3.2.4 Constructing traces for regions

Once the fetching of instructions has begun for a region, the next step is to construct the traces for the region. There are a number of trace construction units available for pre-construction (see Figure 3-5). The regions share the units; with priority based on which region corresponds to the newest start point on the start point stack. The number of trace construction units need not be equal to the number of instruction cache ports available to the pre-construction mechanism. A small worklist is used to maintain trace starting points in a region.

A few initial points in the CFG are identified as starting points of traces. For regions starting at return points, initially only the first instruction of the region is identified as a trace starting point. Because traces end at return instructions, the first trace of the region will start at the first instruction. For regions starting at the exit point of loops, identifying initial starting points is more complicated; when at the time the loop exits, there may be a trace that contains some instructions from the last iteration of the loop and some instructions from beyond the exit of the loop. The trace selection heuristic of forcing traces to end at some even multiple of instructions beyond a backward branch limits the number of initial trace starting points. The initial trace starting points are pushed onto the worklist.

A free trace constructor unit will take a trace starting point from the worklist and attempt to generate all possible traces with that start point. The trace constructor unit fetches the needed instructions from the pre-construction window, requiring one read for each run of contiguous instructions. When a conditional branch is reached, the trace constructor uses the information from the CFG as to which direction is favored. The trace constructor also pushes the decision point onto a small hardware stack so it can generate the alternative trace later. For each new trace constructed, a new trace starting point is identified (corresponding to the instruction following the last instruction of the new trace) and pushed on the worklist. After generating a trace, the trace constructor will pop the last decision point from its hardware stack and back up to start generating the alternative trace.

To avoid tying up trace constructors, a trace start point is not eligible until it is known that all traces starting from it can be generated. Along with each instruction is kept its minimal distance from the first instruction of the region. In the simulated implementation a true minimal distance is maintained, but in a real implementation most likely an approximation would be used. Maintaining the true minimal distance requires occasionally propagating information when a reconvergent path is found. A simple-to-implement approximation is to maintain the distance observed from the start point to reach a current instruction. By looking at the minimal distance of a trace start point and the lowest minimal distance on the worklist, it is easy to make a conservative decision whether all traces from the start point can be constructed.

There is one worklist of trace starting points per active region. The worklists use a simple algorithm to order fetches. The priority of the first identified trace starting points are

one (lower values have higher priority). New trace starting points are generated when a trace is created for an existing trace starting point. The priority value of a new starting point is one higher than the value of the trace starting point that led to it. Trace starting points with equal priority are handled in a first come first serve manner.

3.2.5 The pre-construction buffer

The pre-construction buffer is a small cache accessed in parallel with the trace cache. The pre-construction buffer stores with each trace the start point address of the region that generated the trace. This information is needed for two reasons. First, it is used for determining replacement when a way of the buffer is full. The relative priority of these start points (the newer the start point the higher the priority) is used for determining replacement when a new trace is written to the buffer. Second, the pre-construction engine needs to know when a trace is replaced before it is used. If the region corresponding to a replaced trace is still active, the starting address of the trace is added to the trace construction worklist to be possibly regenerated in the future. If the region corresponding to a replaced trace is in the list of completed regions, the region will be marked as no longer completed.

An important optimization for pre-construction is to make sure there is no redundancy between the trace cache and the pre-construction buffer. This is achieved by checking the trace cache before adding an entry to the pre-construction buffer and checking the pre-construction buffer when adding a trace to the trace cache. If a trace is already in the trace cache, it is not added to the pre-construction buffer. When a trace is added to the trace cache, the copy in the pre-construction buffer, if present, is invalidated.

The pre-construction engine is informed when a trace is replaced in the pre-construction buffer before being used. This information can be used to keep track of which regions may need to have pre-construction performed for them again in the future. This information is a rough approximation because of the interaction between the trace cache and the pre-construction buffer. If a trace is used, it will be placed in the trace cache and removed from the pre-construction buffer. If this trace is subsequently replaced in the trace cache, the pre-construction engine is not informed.

3.3 Results for Length 16 Traces

The result section starts with a summary of the performance of pre-construction. These results give an idea of how a well-tuned pre-construction engine behaves. Next is an analysis of how changes in the pre-construction engine effect performance. Various configurations are tested to demonstrate the range of performance of pre-construction and to determine a well-balanced configuration. For the summary and the analysis of the design space only the two benchmarks *gcc* and *go* are presented. These two benchmarks have the largest instruction working sets of the SPECint95 benchmarks and therefore stress the trace cache the most. Finally, detailed performance results are presented for all the benchmarks. Corresponding results for the SPECfp95 benchmarks are given in Appendix A.

3.3.1 Processor configuration

These results are based on a trace processor that uses traces with a maximum length of 16 instructions. The execution engine is based on the small processor configuration

described in Section 2.3.4. The execution engine consists of four processing elements with 2-way issue per processing elements. This gives the processor an aggregate instruction window size of 64 instructions and a maximum aggregate issue width of eight instructions per cycle. The trace cache is 2-way set-associative and, unless otherwise stated, has 256 lines.

Table 3-1 Default pre-construction configuration.

Parameter	Default Value	Notes
Buffer Size	128 Traces	The number of traces in the pre-construction buffer. The buffer is 2-way set-associative.
Active Regions (Pre-construction windows)	4	The maximum number of active regions. An active region is a region for which the prefetch engine is actively trying to construct traces. An active region is assigned an instruction window and competes for instruction cache ports and trace constructors.
Window Size	256 Insns	The size of the instruction window used to build a CFG for an active region. There is one instruction window per active region.
Instruction Cache Ports	1	The number of instruction cache ports the pre-construction engine can use. We model dedicated ports, although in a real implementation these ports would likely be shared with other activities (I-cache fills and/or fetches for the slow path).
Trace Constructors	4	The number of trace constructors. Trace constructors pull instructions out of the instruction window and create valid traces.
Use Slow path Predictors	True	The pre-construction engine can potentially reference the branch predictor from the slow path hardware to better direct pre-construction.
Bias	3	The drop in priority given to fetches for the less favored (based on prediction from dynamic or static predictor) direction of a conditional branch.
Start point Stack Depth	12	The number of entries in the start point stack. The start point stack is used to keep track of regions for which pre-construction should be performed.
Active Region Buffer Size	16	One active region buffer entry is used for each unique entry in the start point stack. Additional entries are used to remember regions for which pre-construction has recently been completed.

There are many parameters that can be adjusted in the pre-construction engine. The default pre-construction engine configuration is given in Table 3-1. Unless stated otherwise, each parameter will be set to its default value. The work lists for directing fetch and trace construction are sufficiently large as to never fill up.

3.3.2 Example Performance

The reduction in trace cache misses is a good first-cut metric of pre-construction performance. Figure 3-6 and Figure 3-7 give the trace cache miss rates, in the units of misses per 1000 instructions, for a variety of trace cache and pre-construction configurations for the benchmarks *gcc* and *go*, respectively. The graphs present the miss rate as a function of the combined size of the trace cache and the pre-construction buffer. The pre-construction engine will require some additional area, but this is not considered in the comparison. The trace cache size varies over a range of 64 to 1K entries. In section 0 the overall performance implications of reducing the trace miss rate are discussed.

The benchmarks *gcc* and *go* both see significant benefit from trace pre-construction. For a given trace cache size, there is a 30% to 40% decrease in miss rate for the smallest pre-construction configuration up to a 45% to over 50% decrease in miss rate for the largest pre-construction configuration. The benefit from pre-construction is much more significant than allocating comparable area to the trace cache. For comparable area, the best pre-construction configurations offer approximately a 30% to 40% decrease in miss rate for both benchmarks.

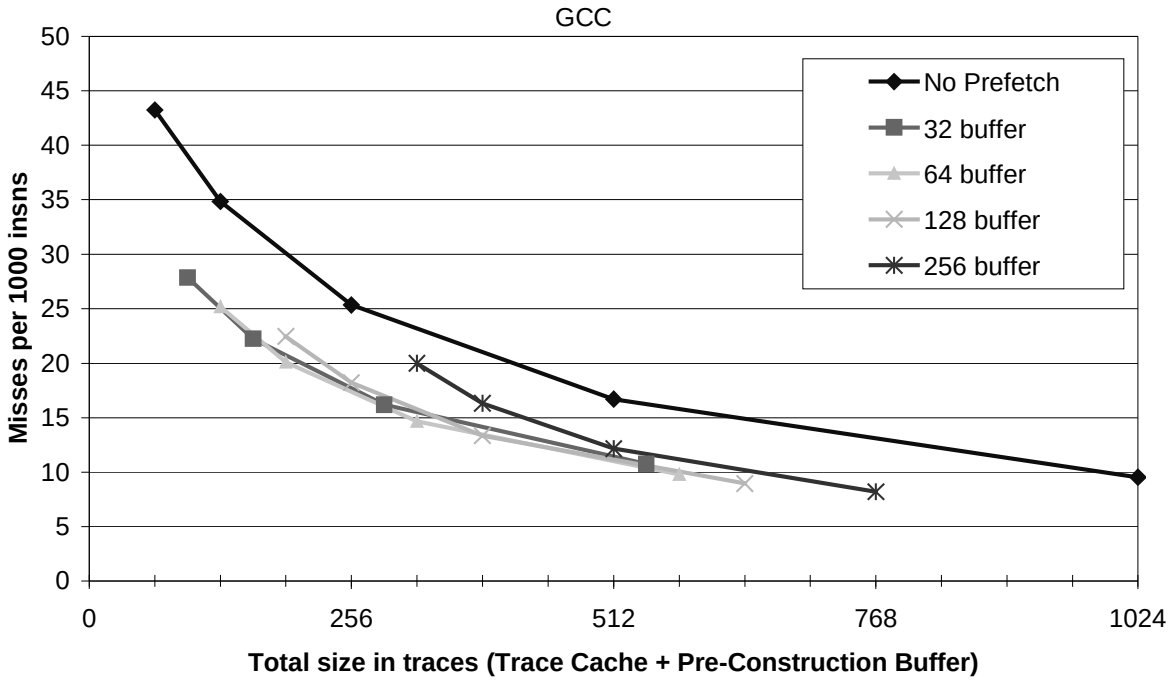


Figure 3-6 Trace miss rate (in misses per 1000 instructions) for *gcc* with 16 long traces.

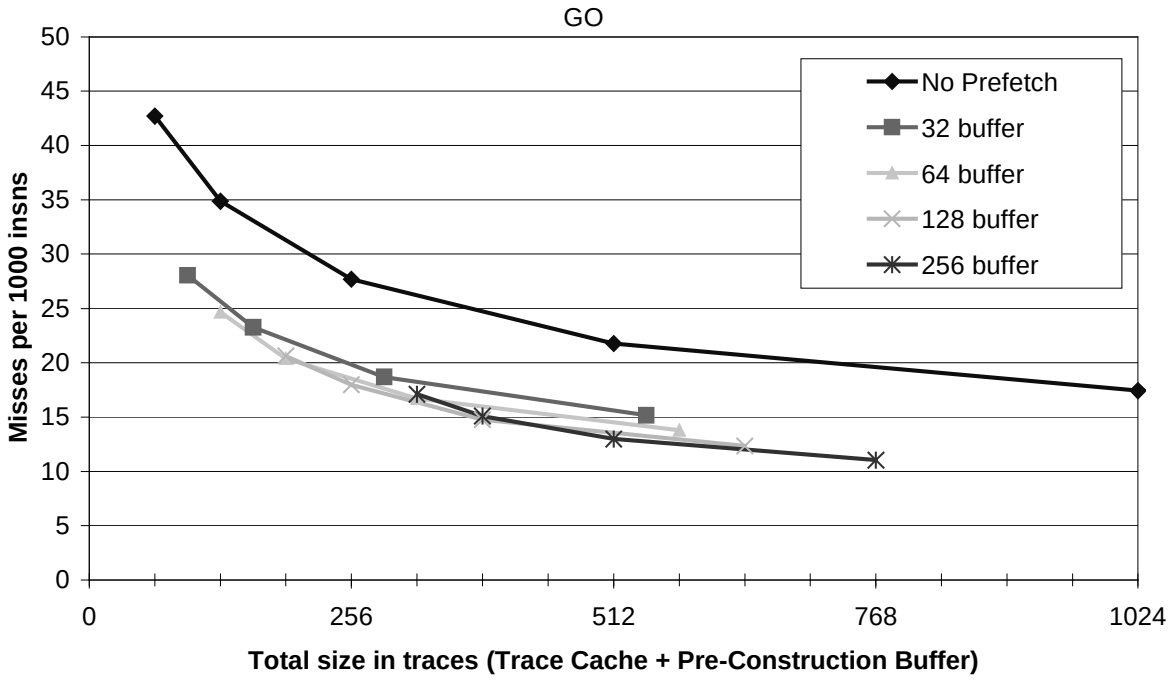


Figure 3-7 Trace miss rate (in misses per 1000 instructions) for *go* with 16 long traces.

The benchmark *gcc* sees the most benefit from incorporating a small pre-construction buffer and allotting most of the area to the trace cache. On the other hand, the benchmark *go* sees the most benefit from a large pre-construction buffer. Because of this behavior, either a compromise has to be made, or a design that dynamically allocates space for the pre-construction buffer needs to be used. A unified buffer (used for both the trace cache and the pre-construction buffer) would offer higher performance, but would likely be very complex to implement. I do not investigate a unified buffer approach further. The tradeoffs of a unified buffer are similar to the tradeoff of having separate instruction and data caches or a single unified cache. Unified (instruction and data) caches offer higher potential performance, but most processor designs use separate caches to support higher bandwidth and enable the caches to be customized for different tasks.

3.3.3 Tuning the Configuration

3.3.3.1 Number of active regions and depth of region start point stack

Adding more active regions to the pre-construction engine has two effects. First, work can be performed on multiple regions in parallel. The amount of parallelism is dependent on the resources available. In practice, the highest priority region uses most of the resources. The second, and more significant, benefit of multiple active regions is to preserve the work done on a region after a higher priority region is started. This enables the pre-construction engine to continue a region after the other region completes or is squashed.

There is a significant benefit to adding more active regions up to four, beyond four there is minimal benefit and even some degradation when adding more regions (see Figure 3-8 and

Figure 3-9). This performance curve reflects the behavior of loops and subroutines. Consider the case where the pre-construction engine is working on the region following a call to subroutine *A* and a nested call within *A* to subroutine *B* is observed. There is a good opportunity, if there are enough active regions, to perform pre-construction on the portion of subroutine *A* following the call and preserve the work from the earlier observed call. Beyond a point, remembering regions corresponding to calls or loop edges observed in the past can become detrimental as subroutines and loops tend to be either very small or very large. If there is a significant amount of code seen after a call or back edge, there is likely to be even more code executed before the corresponding return or loop exit is seen. The work for such a region therefore may be of little use, as it will be ultimately displaced before it can be used; it also may displace other potentially useful information in the pre-construction buffer.

The pre-construction engine has the highest performance with 4 active regions, but the lower cost implementation with only two regions is worth noting. The pre-construction engine performs relatively well with only 2 active regions. Depending on the cost of implementation, this smaller implementation may offer a better price/performance ratio.

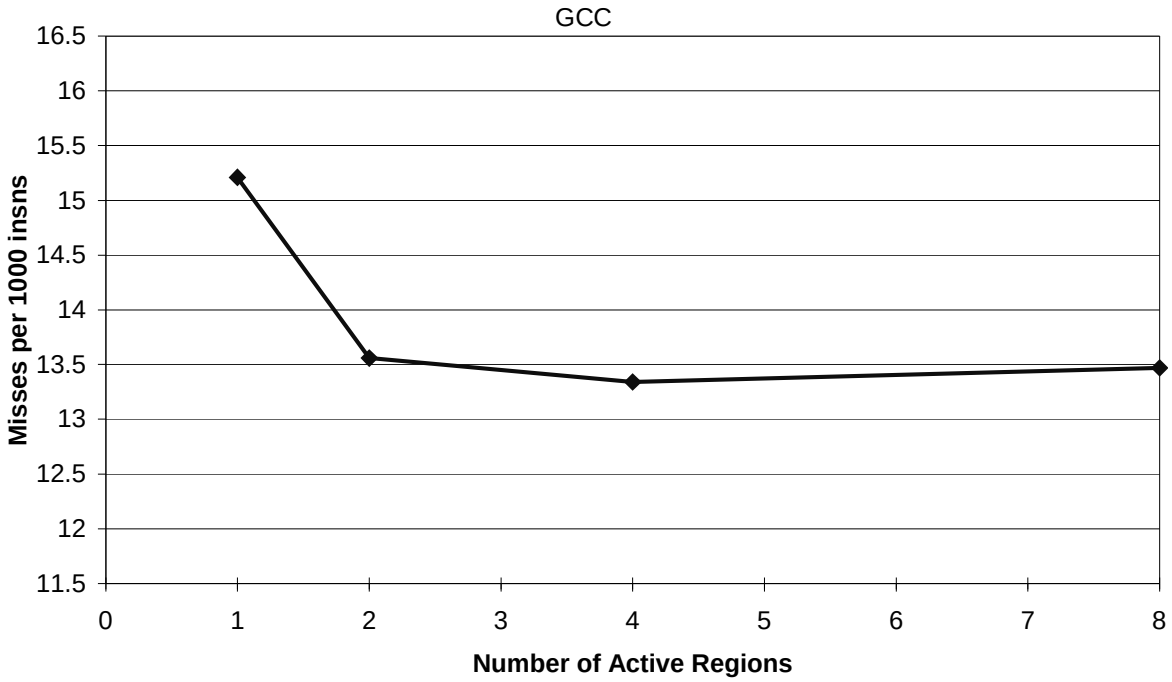


Figure 3-8 Effect of changing the number of active regions for *gcc*.

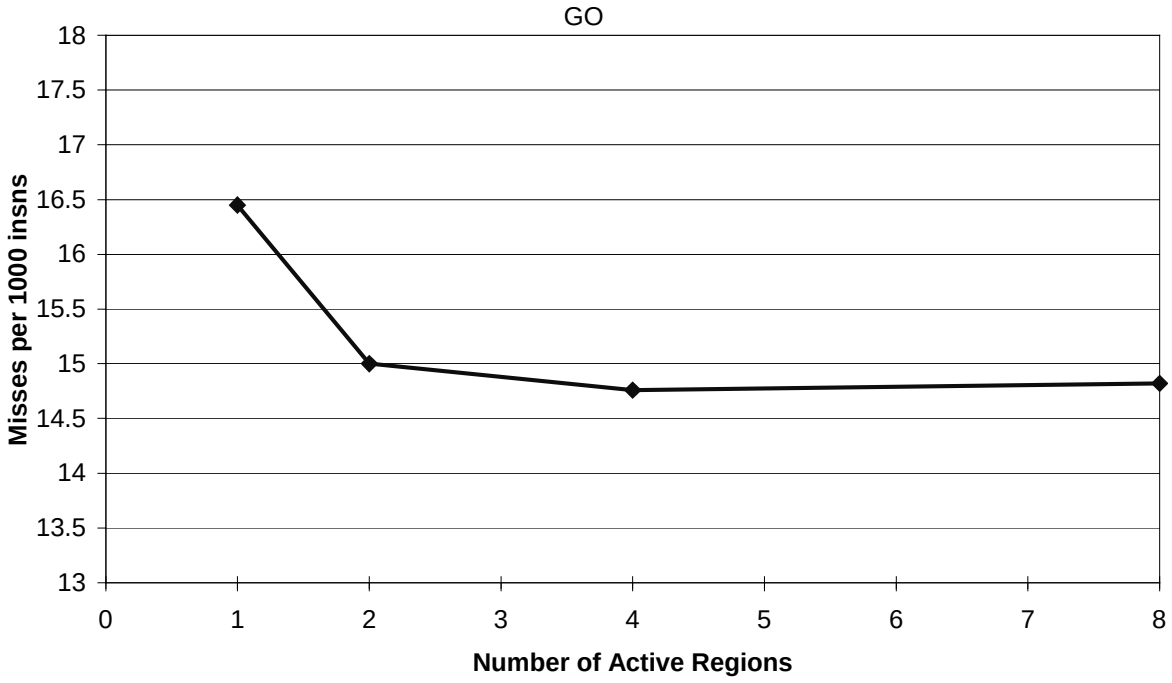


Figure 3-9 Effect of changing the number of active regions for *go*.

The start point stack and active region buffer are closely tied together. The start point stack must be smaller than the active region buffer. The performance impact of altering the size of the active region buffer is shown in Figure 3-10 and Figure 3-11 (for a buffer size of 8 a start point stack depth of 4 is used, for larger buffer sizes a start point stack depth of 12 is used). Adding more entries to the active region buffer allows the pre-construction engine to remember more regions that have been completed to avoid redundant pre-construction. It can be detrimental to remember for too long a period of time that a specific region completed. The only way a completed region transitions to not completed is when the buffer overflows or if a trace corresponding to the region is replaced before being used. If a trace is used, and therefore brought into the trace cache, the region will still be considered completed even if the trace is subsequently replaced in the trace cache. The results suggest that a size of around 16 is best for the active region buffer.

The depth of the start point stack cannot be increased without also increasing the size of the active region buffer. To identify the impact of a shorter stack depth, a stack with fewer entries was tested while keeping the active region buffer at 16. To identify the impact of a deeper stack depth, a stack depth with more than 12 entries was tested, increasing the size of the active region buffer comparably. The results from these tests are shown in Figure 3-12 and Figure 3-13. For stack depths less than twelve, the trace cache miss rate goes up. For deeper stack depths, the miss rate goes up comparable to increasing the size of the active region buffer alone. This suggests that there is little or no benefit from a deeper stack.

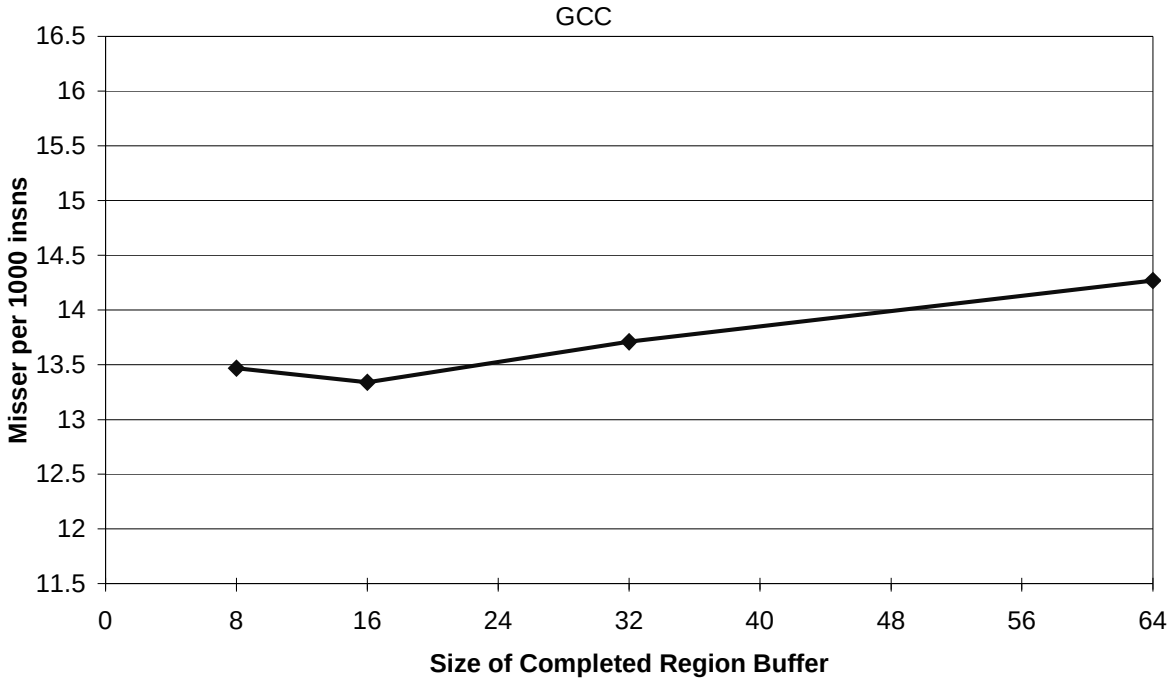


Figure 3-10 Effect of changing the size of the active region buffer for *gcc*.

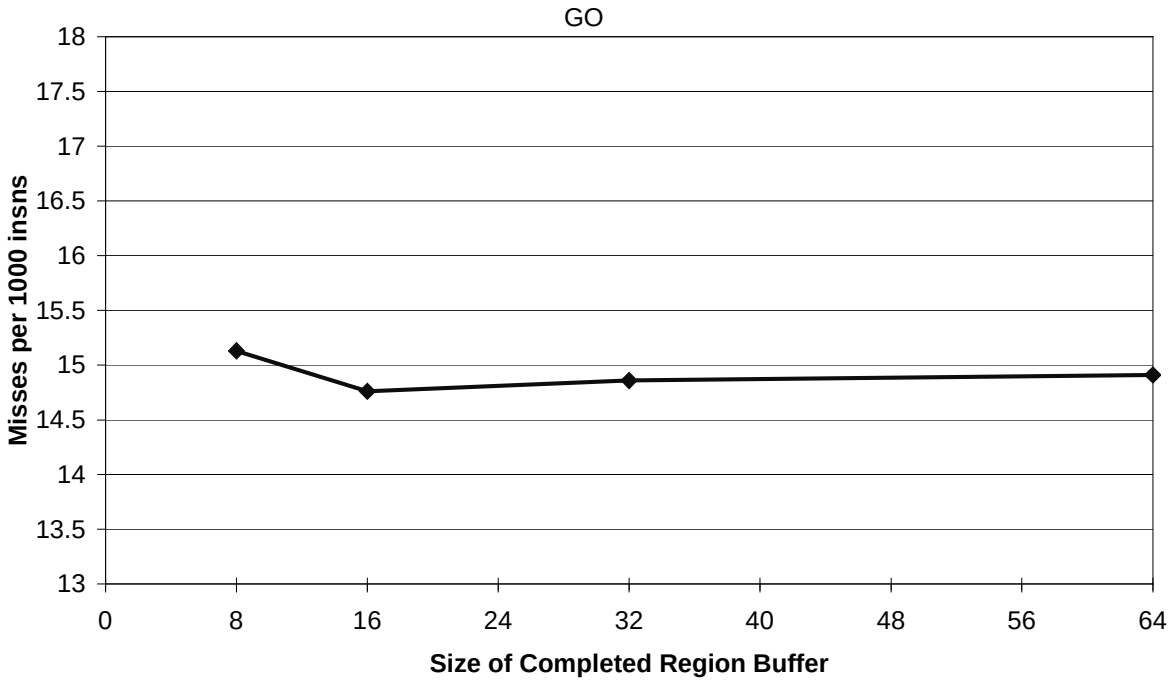


Figure 3-11 Effect of changing the size of the active region buffer for *go*.

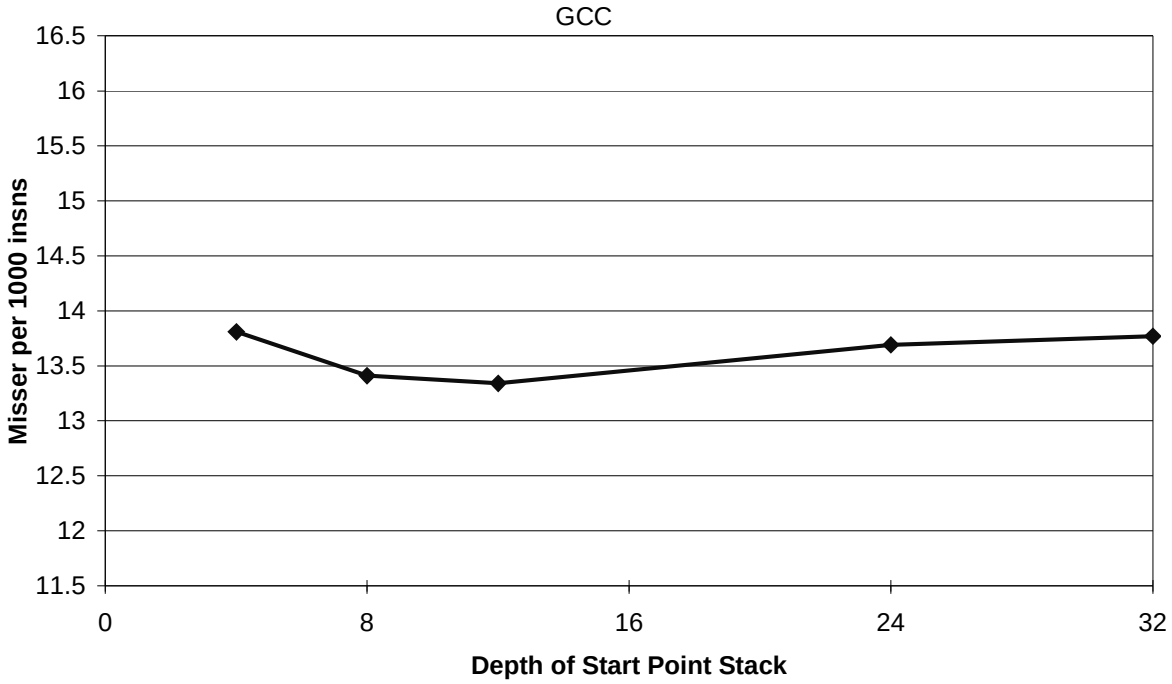


Figure 3-12 Effect of changing the depth of the start point stack for *gcc*.

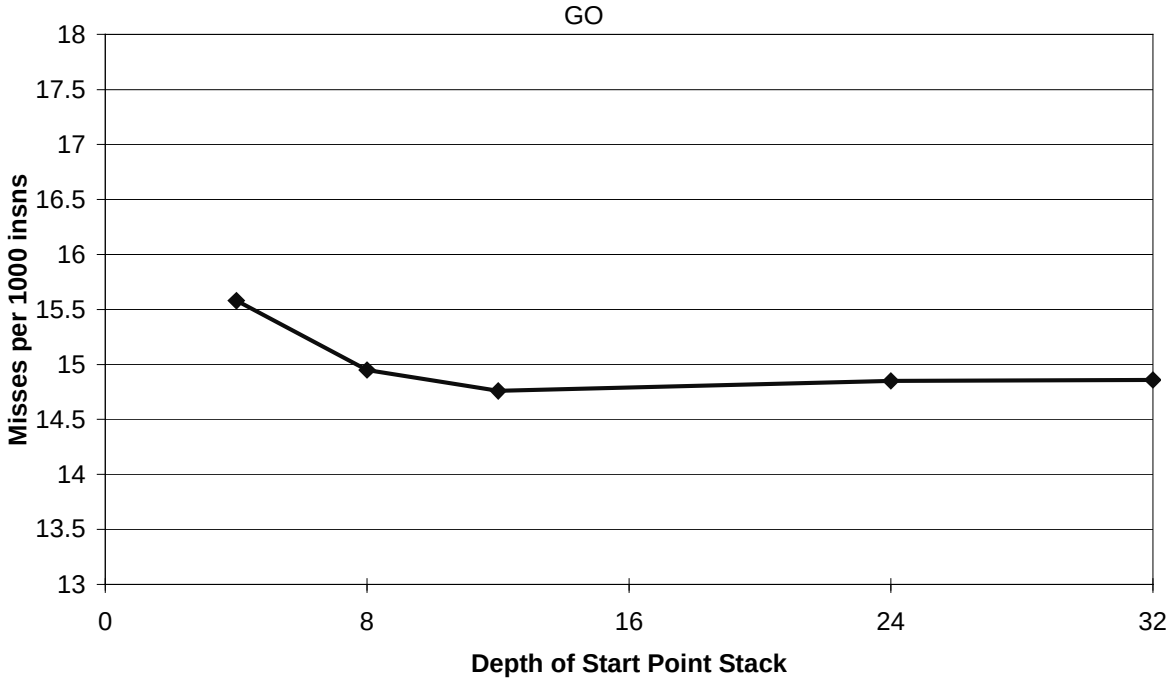


Figure 3-13 Effect of changing the depth of the start point stack for *go*.

3.3.3.2 Instruction cache ports and trace constructors

The most valuable resource used by the pre-construction engine is probably instruction cache ports, as adding additional ports to a cache may significantly impact its size and latency. The number of instruction cache ports available to the pre-construction engine determines the instruction fetch bandwidth of the engine. Balanced with the instruction cache ports are the trace constructors, which consume the instructions. Trace constructors require a non-trivial amount of hardware, although their cost is still relatively small. Figure 3-14 and Figure 3-15 show the affects of varying the number of instruction cache ports and the number of trace constructors.

There is a significant performance benefit when going from one instruction cache port to two. Going beyond two instruction cache ports does not seem to have much additive benefit. If two cache ports were available, it would make sense to let the pre-construction engine use them both. However, the performance benefit of a second instruction cache port probably does not justify the cost of adding a port if it is not already available for other reasons. The benefit of more instruction cache ports is the increase in bandwidth. Other approaches to increase the fetch bandwidth, such as using longer cache lines, would likely provide some of the same benefit of additional ports.

Adding additional trace constructors leads to notable performance improvements up to four trace constructors. Beyond four, the benefit of additional trace constructors is minimal. For a pre-construction engine with two instruction cache ports, it would make sense to have four trace constructors. For a pre-construction engine with only one instruction cache port,

four trace constructors probably still represent the best balance (making the best use of the high cost instruction cache port). A lower cost implementation with only one port and two trace constructors also performs well.

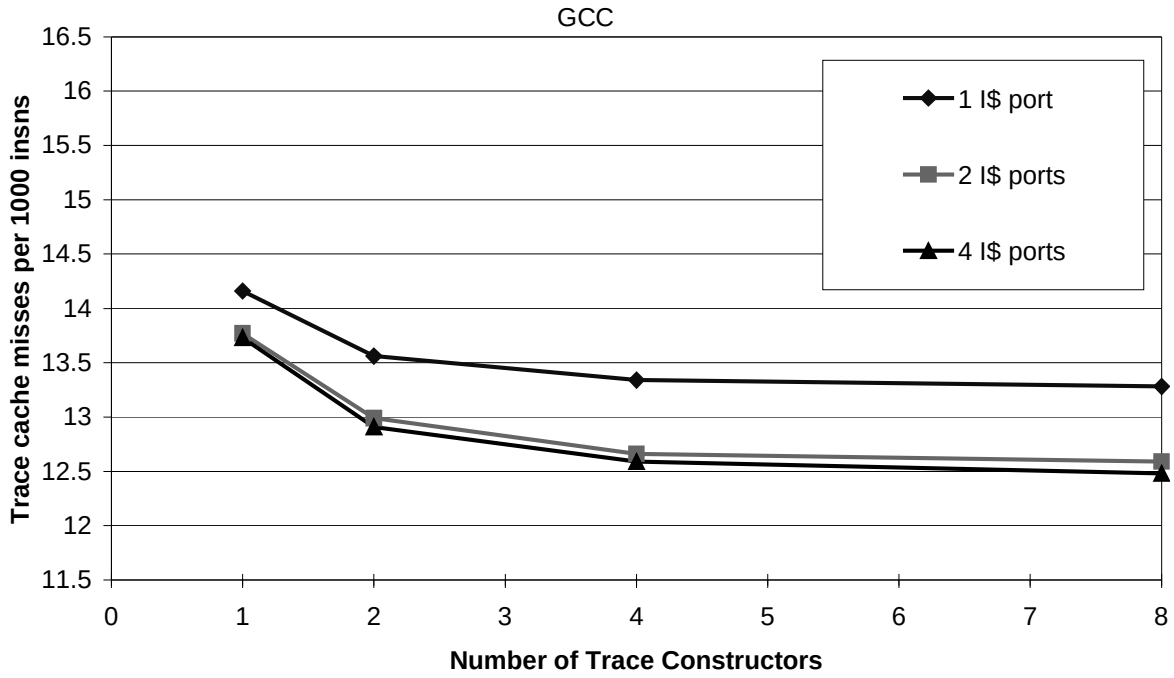


Figure 3-14 Effect of changing the number of I-cache ports and trace constructors for *gcc*.

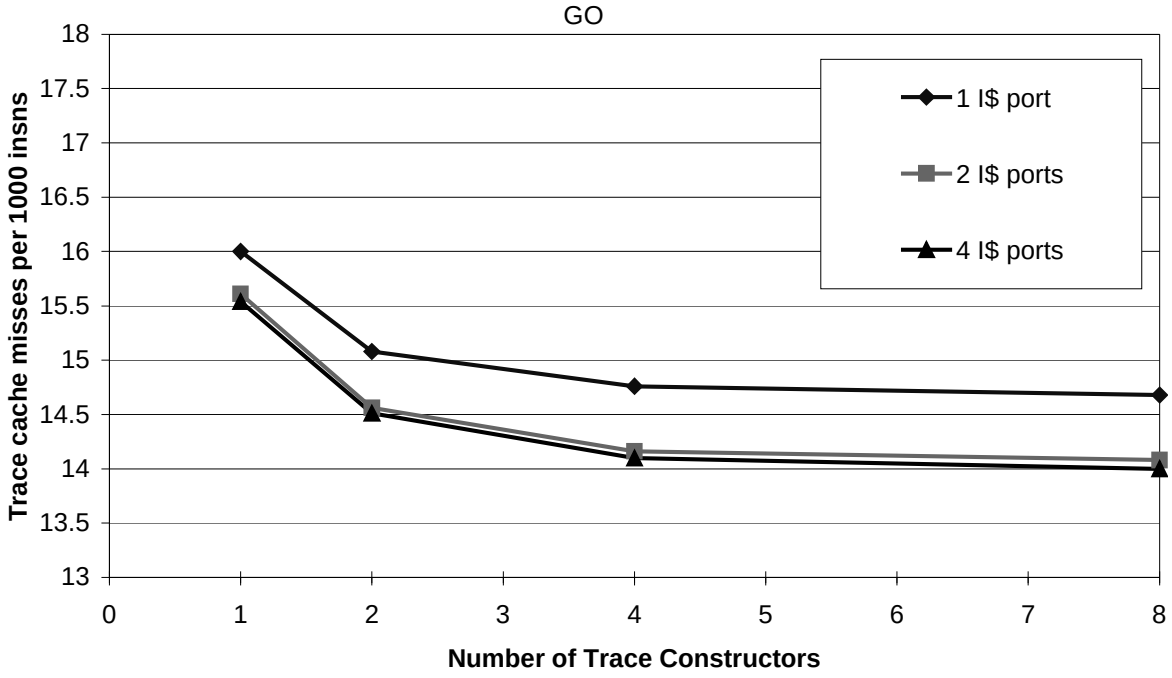


Figure 3-15 Effect of changing the number of I-cache ports and trace constructors for *go*.

3.3.3.3 Pre-construction window size

The size of the pre-construction window has a significant impact on the performance of pre-construction. If the pre-construction window is not large enough, the pre-construction engine can not capture a significant portion of each region. If the instruction window is too large, the pre-construction engine may spend too much time on a single region at the expense of other regions. A window size around 256 instructions appears to be the best (see Figure 3-16 and Figure 3-17). This result may partially be an artifact of the compiler and the workload. This size of window likely corresponds to the common size of constructs (such as subroutines and loop bodies) in the applications. The sensitivity of the pre-construction engine to the window size is likely also dependent on the bandwidth of the resources to fill and drain the window.

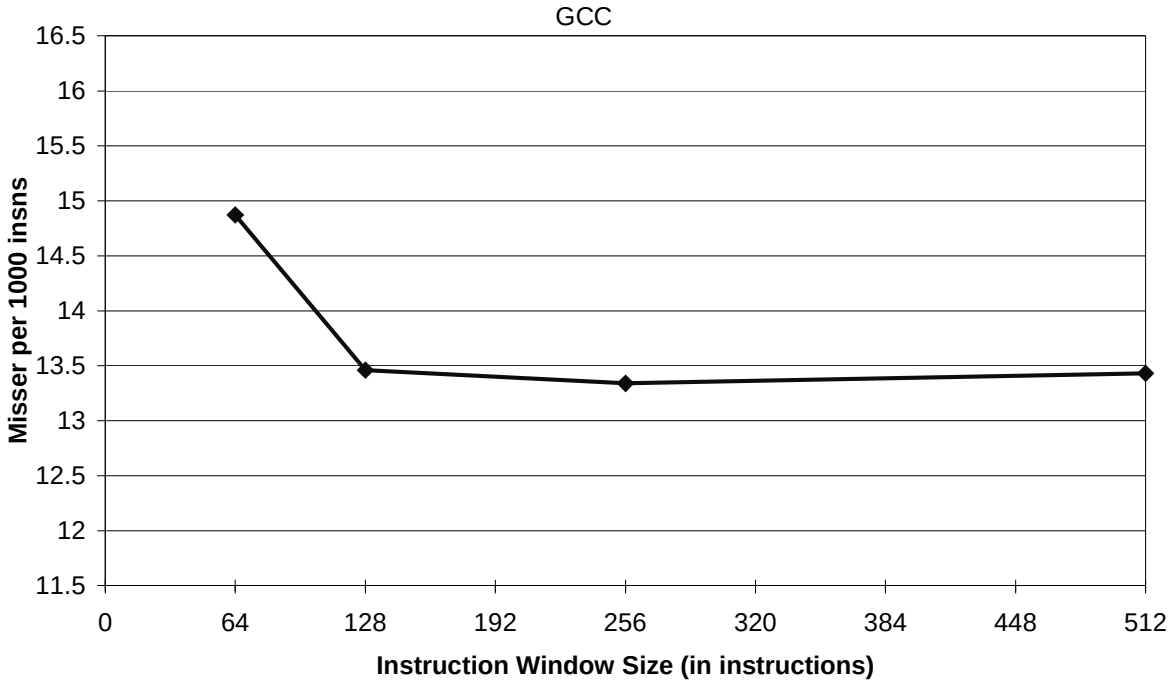


Figure 3-16 Effect of changing the window size for *gcc*.

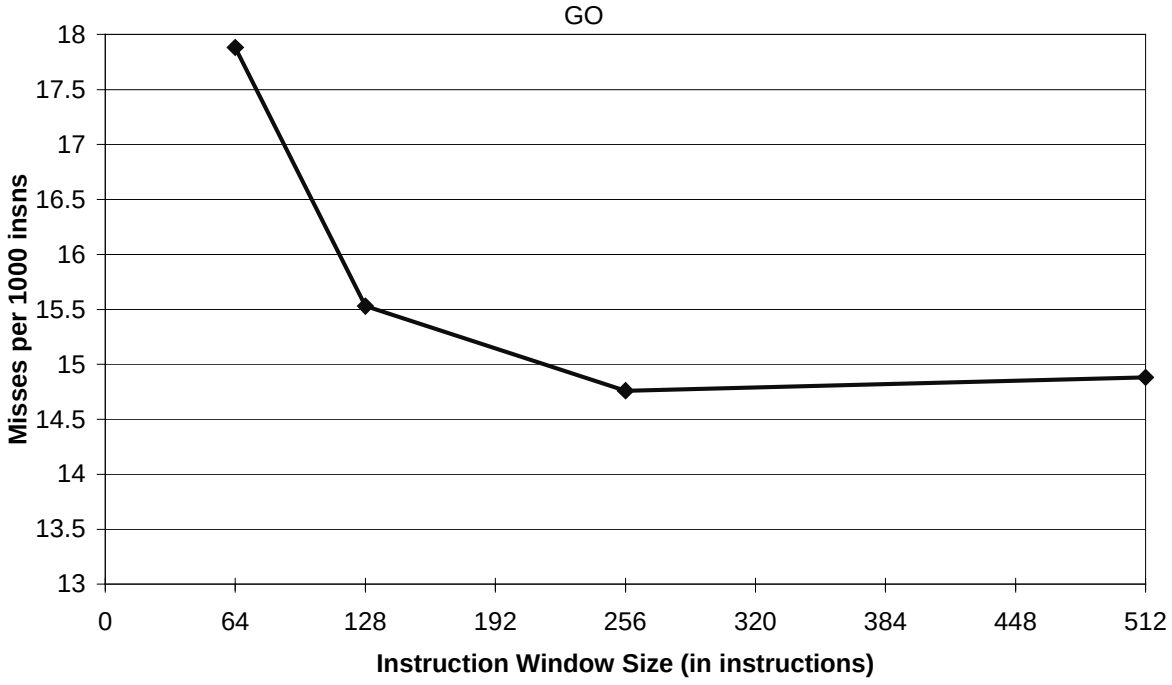


Figure 3-17 Effect of changing the window size for *go*.

3.3.3.4 Utilizing the branch predictor

The branch predictor in the slow path is referenced for branches fetched in the pre-construction engine. The prediction is used for two reasons. First, it determines if a branch is highly biased. If a branch is highly biased it may be better to only fetch for the favored path. Second, the favored direction of a branch can be used to determine the order instructions are fetched and the order that traces are constructed. Forward branches commonly correspond to if-then-else constructs, and for these branches a strong bias in either the taken or not taken direction is considered. Backward branches commonly correspond to loop constructs. Backward branches that are highly biased not-taken likely correspond to loops that are never taken and should be considered for directing fetch. Backward branches that are highly biased in the taken direction or weekly biased in either direction are considered to be loops that are taken for some number of iterations and then exited. For these branches both directions should be followed.

Referencing the branch predictor to help direct pre-construction leads to notably better performance (see Figure 3-18 and Figure 3-19). The affect of varying the bias between the favored and non-favored directions of branches is not significant. A bias of three performs at least as well as any other (see Figure 3-18 and Figure 3-19). Because the bias of one performs nearly as well as a higher bias, the direction prediction of branches is not significant. What is significant is identifying branches that are highly biased so that only the dominant direction is followed.

If the compiler can provide information identifying the majority of highly biased branches, then referencing the branch predictor would not be necessary. The compiler could

likely identify many of these branches. The cost of referencing the branch predictor is a couple of extra read ports for the prediction table, which is a non-trivial cost. A design that could avoid this extra cost would be desirable.

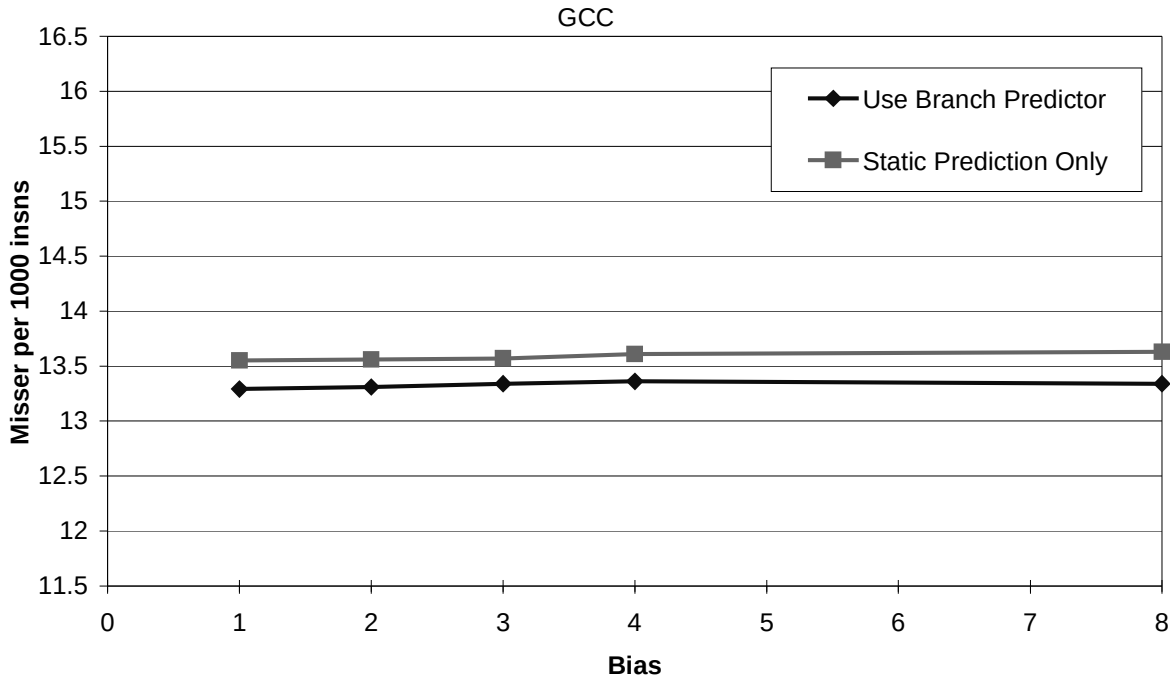


Figure 3-18 Effect of using dynamic branch prediction and various biases for *gcc*.

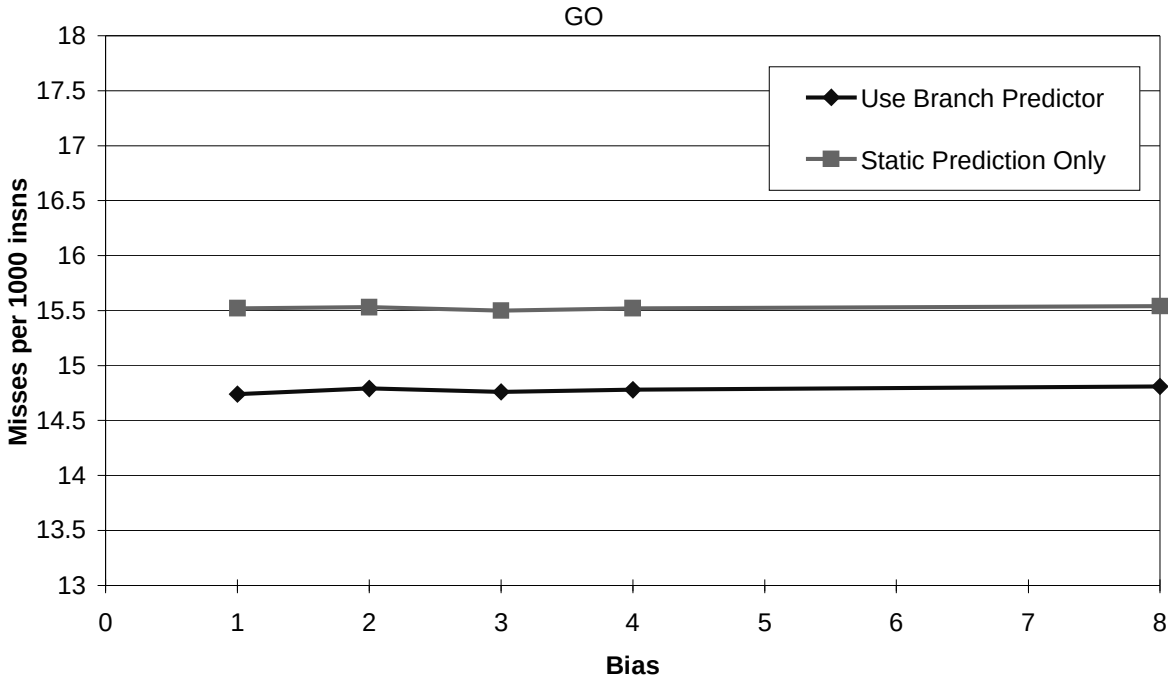


Figure 3-19 Effect of using dynamic branch prediction and various biases for *go*.

3.3.4 Trace cache miss rate

In section 3.3.2 the trace cache performance of the two benchmarks *gcc* and *go* is given. In this section the trace cache performance of the remainder of the SPECint95 benchmarks is given (results for the SPECfp95 benchmarks are presented in Appendix A). The performance is given in terms of the number of trace cache misses per 1000 instructions. The performance is given as a function of the combined size of the trace cache and the pre-construction buffer. Only pre-construction buffer sizes of 32 and 64 traces are considered. In most cases only trace cache sizes through 256 entries will be considered. Because of the small working sets of these benchmarks, the performance of larger trace caches quickly approach the performance of an ideal trace cache.

Figure 3-20 through Figure 3-25 show the performance of the trace cache and pre-construction engine for the remainder of the SPECint95 benchmarks. Two of these benchmarks, *compress* and *ijpeg*, have such small working sets that even a very small trace cache performs well and there is little opportunity to improve. The benchmarks *lisp, m88ksim* and *perl* all show notable benefits with pre-construction. Of these, *perl* stresses the trace cache the most and therefore has the largest absolute gains. The benchmark *vortex* stresses the trace cache almost as much as *gcc* or *go*. Pre-constructions works extremely well for *vortex*, reducing the miss rate by up to a factor of five (see Figure 3-25).

All the benchmarks, except for *compress*, see a lower trace cache miss rate with pre-construction. *compress* sees a minor increase in trace cache miss rate with some configurations. The increase is due the trace cache being reduced in size as area is dedicated to the pre-construction buffer. A point where equal area configurations can be compared across all the benchmarks is a total trace cache size of 128 lines (comparing a 128 entry trace cache with a 64 entry trace cache and a pre-construction engine with a 64 entry pre-construction buffer). The average reduction in trace cache misses is 15% (39% if *compress* is excluded). The harmonic mean of the trace cache miss reduction (excluding *compress*) is 34%.

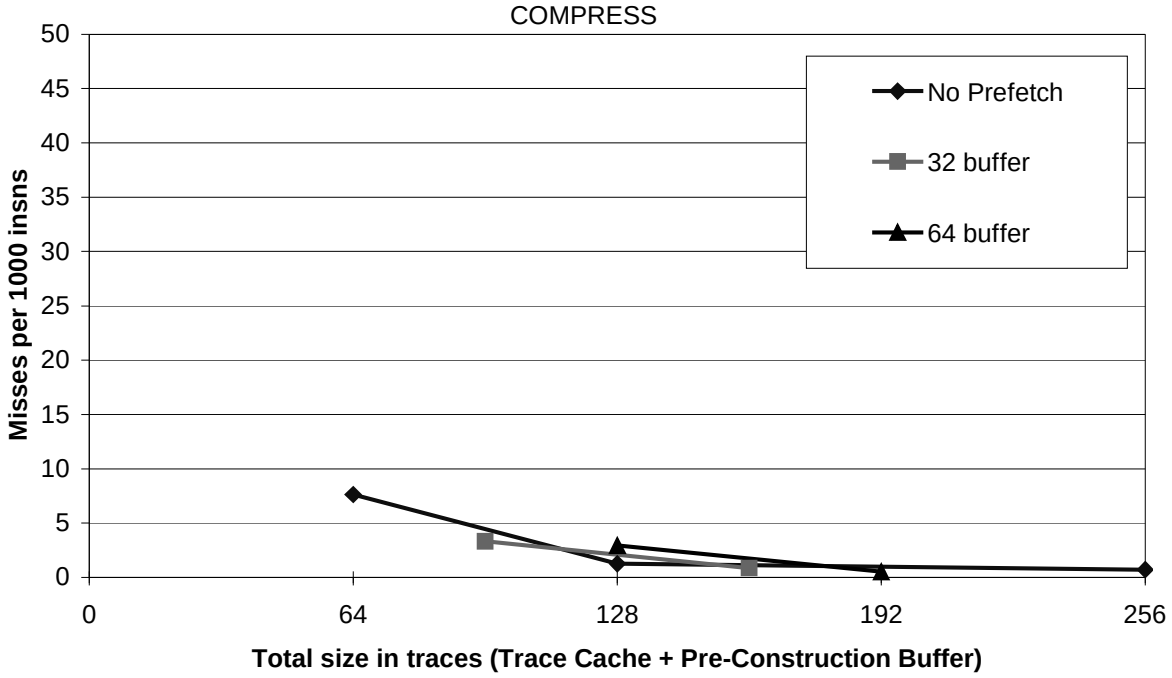


Figure 3-20 Trace cache performance for *compress*.

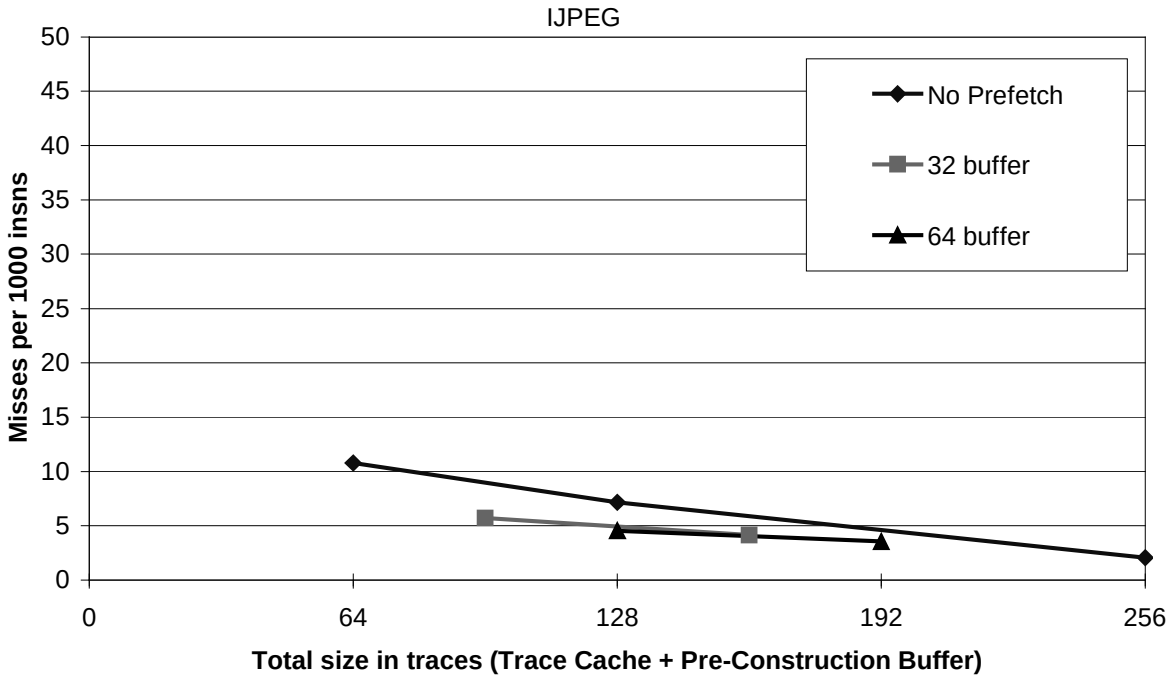


Figure 3-21 Trace cache performance for *ijpeg*.

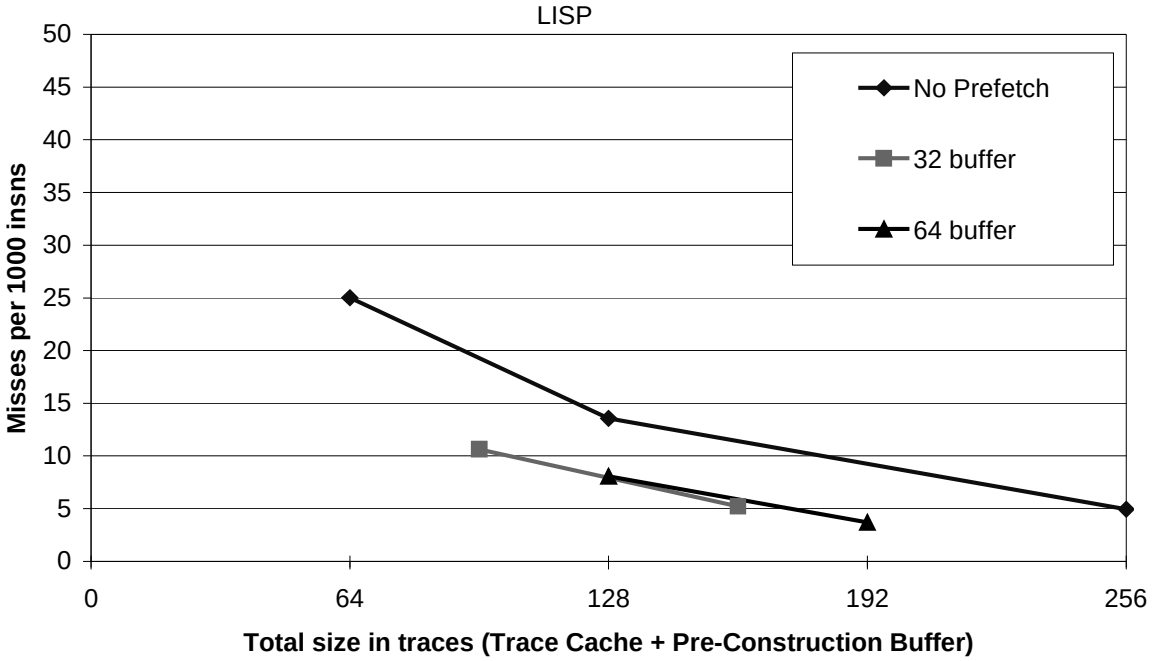


Figure 3-22 Trace cache performance for *lisp*.

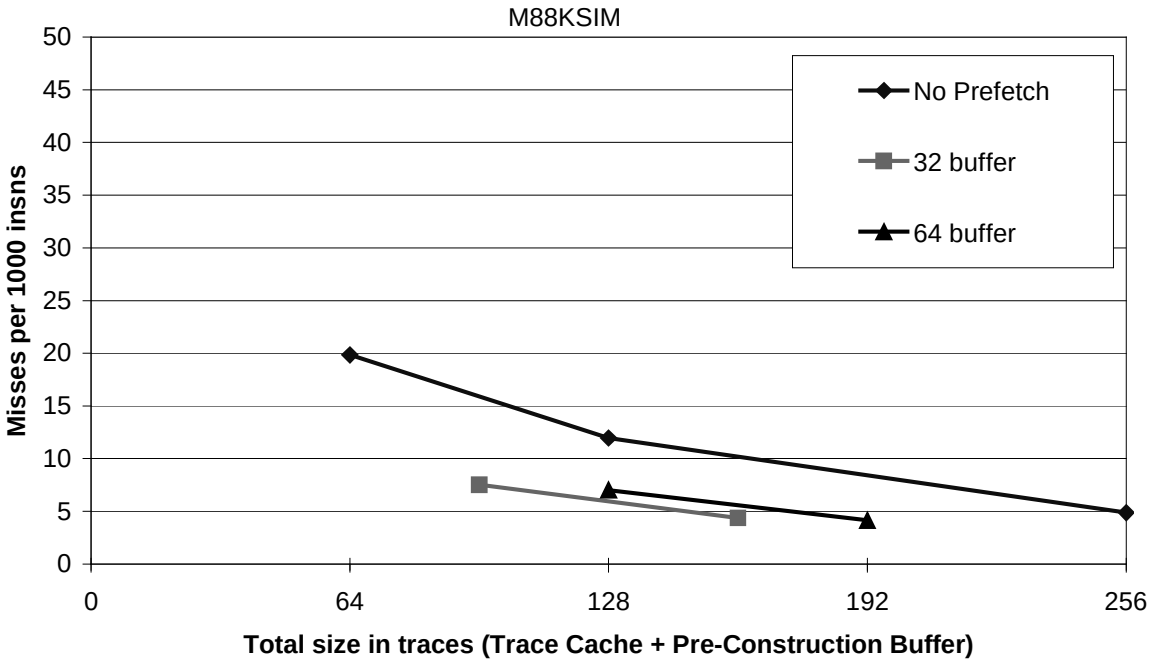


Figure 3-23 Trace cache performance for *m88ksim*.

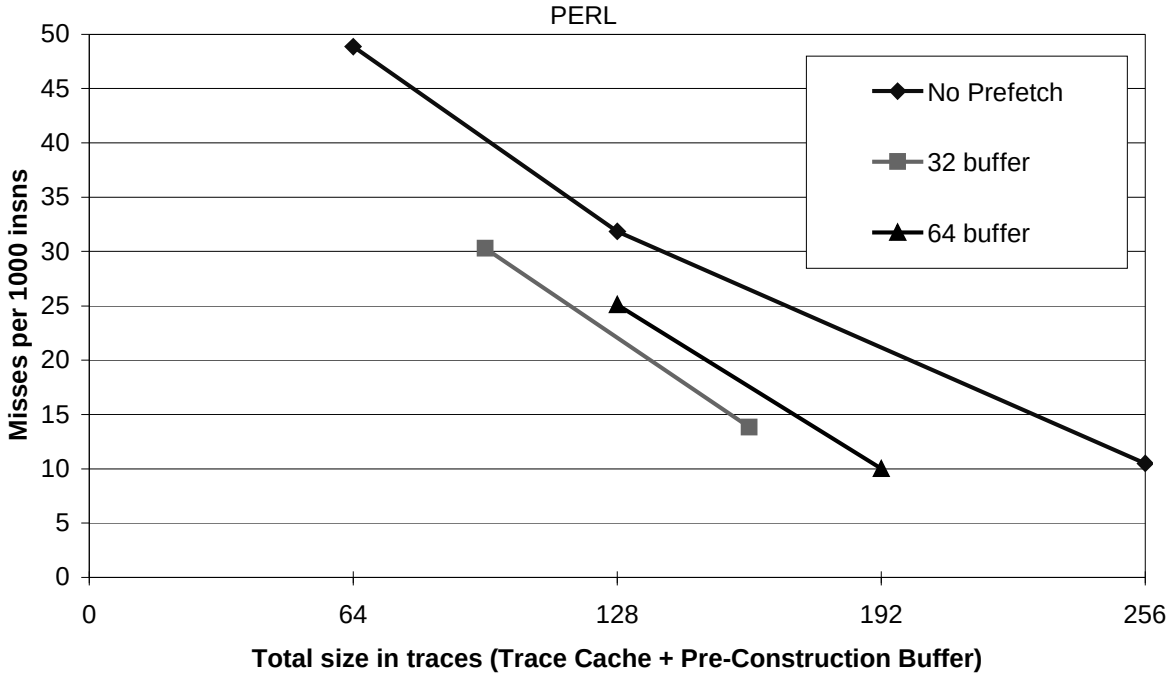


Figure 3-24 Trace cache performance for *perl*.

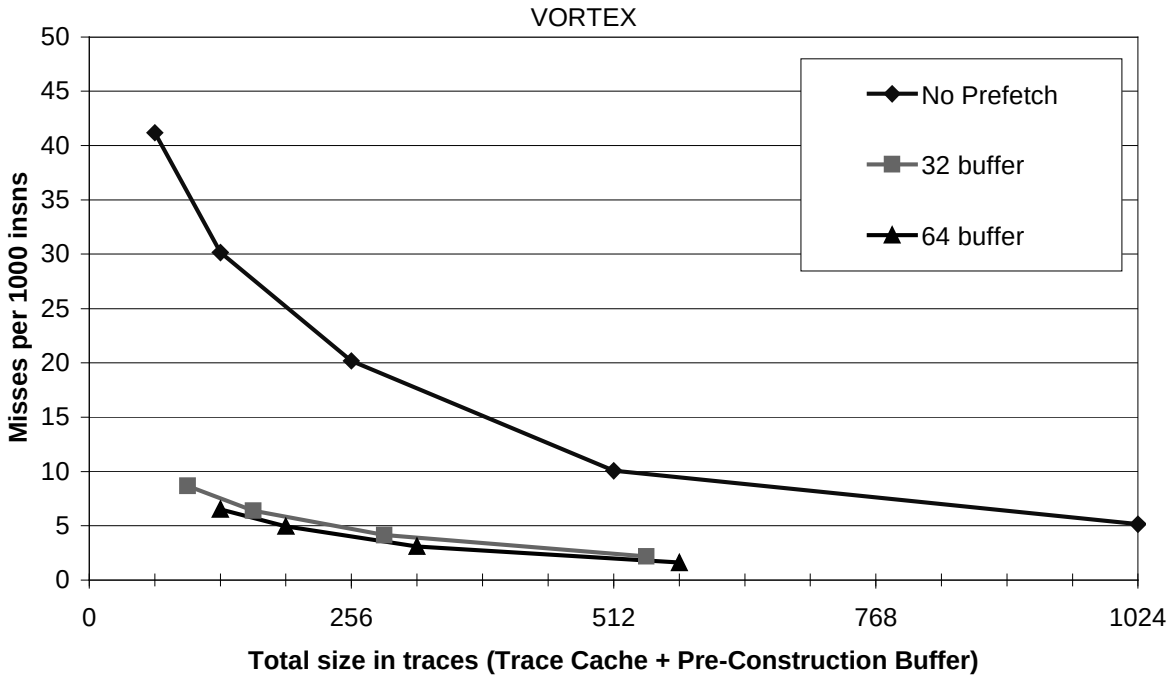


Figure 3-25 Trace cache performance for *vortex*.

3.3.5 Impact on instruction cache performance

Pre-construction has a number of important impacts on the performance of the instruction cache. By changing the trace cache miss rate, pre-construction affects the number of instructions supplied by the instruction cache to the decode pipeline. At the same time, pre-construction increases the number of overall requests to the instruction cache by generating requests itself. Last, but not least, the pre-construction actually prefetches lines into the instruction cache.

By increasing the number of trace cache hits, pre-construction reduces the number of instructions that need to be supplied from the instruction cache. Table 3-2 shows the number of instructions that are fetched from the instruction cache for the benchmarks with and without pre-construction. All the benchmarks, except for *compress*, see over a 20% reduction in the number of instructions supplied from the instruction cache.

Table 3-2 Instructions supplied by the instruction cache (slow path).

Benchmark	**Instruction supplied by I-cache with 128 (512 for *gcc* & *go*) entry trace cache (instructions per 1000 instructions)**	**Instruction supplied by I-cache with 64 (256 for gcc & go) entry trace cache and 64 (256 for *gcc* & *go*) entry pre-construct buffer (instructions per 1000 instructions)**
compress	37	67
gcc	233	181
go	326	213
ijpeg	124	86
li	160	107
m88ksim	154	106
perl	435	337
vortex	416	93

The potential drawback of any prefetching scheme is an increase in memory traffic. Pre-construction requires large bandwidth from the instruction cache, but this does not interfere with other memory requests. Pre-construction may also increase the number of instruction cache misses that are issued to lower-levels of memory. These instruction cache misses will compete with other memory requests, so quantifying the increase is important. Table 3-3 gives the instruction cache miss rates with and without pre-construction. All the benchmarks, except for *m88ksim* (which sees a larger increase), see between a 50% and 100% increase in instruction cache misses. The increase comes from fetches generated by the pre-construction engine for regions of code that are never executed.

Table 3-3 Increases to instruction cache misses from pre-construction.

Benchmark	**Instruction cache misses with 128 (512 for *gcc* & *go*) entry trace cache (misses per 1000 instructions)**	**Instruction cache misses with 64 (256 for *gcc* & *go*) entry trace cache and 256 entry pre-construct buffer (misses per 1000 instructions)**
compress	0.002	0.003
gcc	3.0	6.2
go	7.8	11
ijpeg	0.03	.05
li	0.004	.006
m88ksim	0.01	.13
perl	0.008	.02
vortex	1.5	3.1

Pre-construction increases the total number of instruction cache misses, but it reduces the number of instructions cache misses observed by the slow path. Table 3-4 shows the number of instructions supplied from instruction cache misses with and without pre-construction. Part of the reduction is due to fewer instructions being supplied by the instruction cache.

But, the reduction in instructions supplied from instruction cache misses is greater than the reduction in total instructions supplied from the instruction cache. This is primarily seen in the case of *gcc* and *go*. This suggests that the pre-construction engine is prefetching instruction cache lines that are used by the slow path fetch mechanism.

Table 3-4 Instruction cache misses observed by slow path.

Benchmark	**Instructions supplied from I-Cache miss with 128 (512 for *gcc* & *go*) entry trace cache (instructions per 1000 instructions)**	**Instructions supplied from I-Cache miss with 64 (256 for *gcc* & *go*) entry trace cache and 64 (256 for *gcc* & *go*) entry pre-construct buffer (instructions per 1000 instructions)**
compress	0.007	0.002
gcc	10	7.1
go	35	14
ijpeg	0.13	0.06
li	0.01	0.005
m88ksim	0.03	0.03
perl	0.02	0.01
vortex	7.4	1.8

The instruction cache prefetching effect of pre-construction can be more clearly seen by considering a degenerate case of pre-construction. Using a pre-construction engine that performs the task of fetching instructions for performing pre-construction, but never actually construct traces, isolates the instruction prefetch contribution. The number of instructions supplied from instruction cache misses for this degenerate case is shown in Table 3-5.

Table 3-5 Instruction cache misses observed with degenerate case of pre-construction.

Benchmark	Instructions supplied from I-Cache miss with 128 (512 for *gcc* & *go*) entry trace cache (instructions per 1000 instructions)	Instructions supplied from I-Cache miss with 128 (512 for *gcc* & *go*) entry trace cache and degenerate case of pre-construction (instructions per 1000 instructions)
compress	0.007	0.002
gcc	10	7.0
go	35	
ijpeg	0.13	0.06
li	0.01	0.005
m88ksim	0.03	0.03
perl	0.02	0.01
vortex	7.4	1.9

3.3.6 Impact on overall performance

The real measure of any performance optimization is how much it reduces the execution time. Figure 3-26 through Figure 3-33 show the performance improvements for the SPECint95 benchmarks. For these benchmarks the performance benefit of adding pre-construction is between 0% and 10%. For a total trace cache size (trace cache and pre-construction buffer) of 128 entries the average speedup is 3% with a harmonic mean of 1.5% (*compress* is excluded from the harmonic mean). For the three benchmarks with the highest trace cache miss rate, *gcc*, *go* and *vortex*, the average speedup is 5%. This is a modest, but respectable, performance improvement. By combining pre-construction with the pre-processing optimization the performance improvement is even more significant (this is shown in Section 4.6).

The performance improvement from pre-construction comes from both reducing the number of trace cache misses and reducing the demand instruction cache misses. To break down the improvement, the graphs show the benefit of the instruction cache prefetching

performed by the pre-construction engine. The performance of "instruction cache prefetching" is based on a degenerate pre-construction engine that fetches instructions for pre-construction regions, but does not generate traces. For these results the size of the pre-construction buffer is zero. *gcc* sees about one third of the total benefit come from instruction cache prefetching, while *go* sees over one half of the total benefit come from instruction cache prefetching. The rest of the benchmarks see minimal benefits from instruction cache prefetching.

By reducing the trace cache miss rate, pre-construction increases the peak rate at which instructions can be fetched into the instruction window. The focus is the peak fetch bandwidth, not the average fetch bandwidth. The average instruction fetch rate can not be higher than the number of instructions retired per cycle, which is less than a basic block per cycle. Trace caches (and pre-construction) help performance by filling the instruction window quickly, to expose potentially independent instructions, when the window is nearly empty. A nearly empty instruction window is most commonly caused by control mispredictions which force a significant part of the window to be flushed.

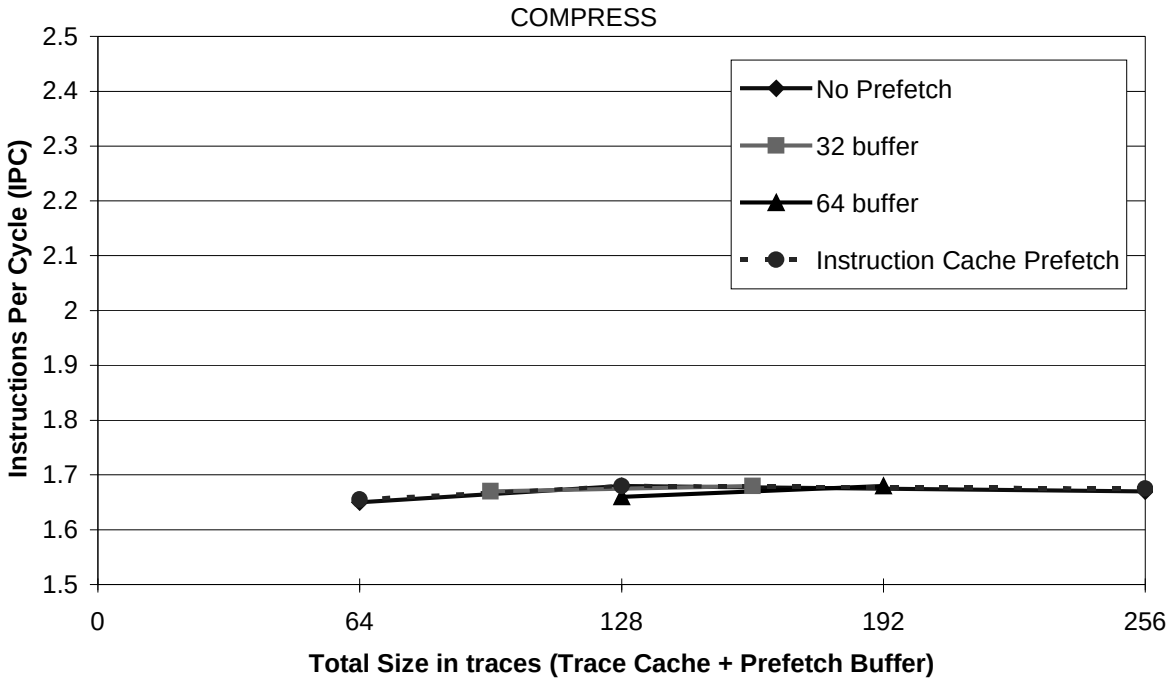


Figure 3-26 Performance for *compress.*

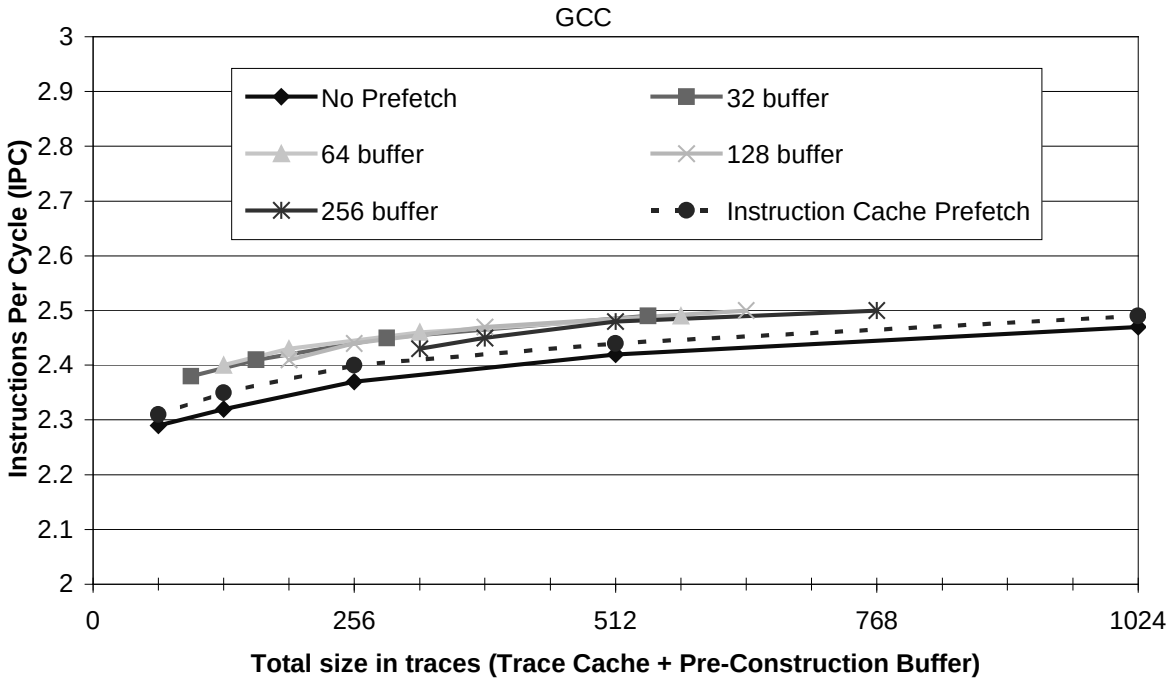


Figure 3-27 Performance for *gcc*.

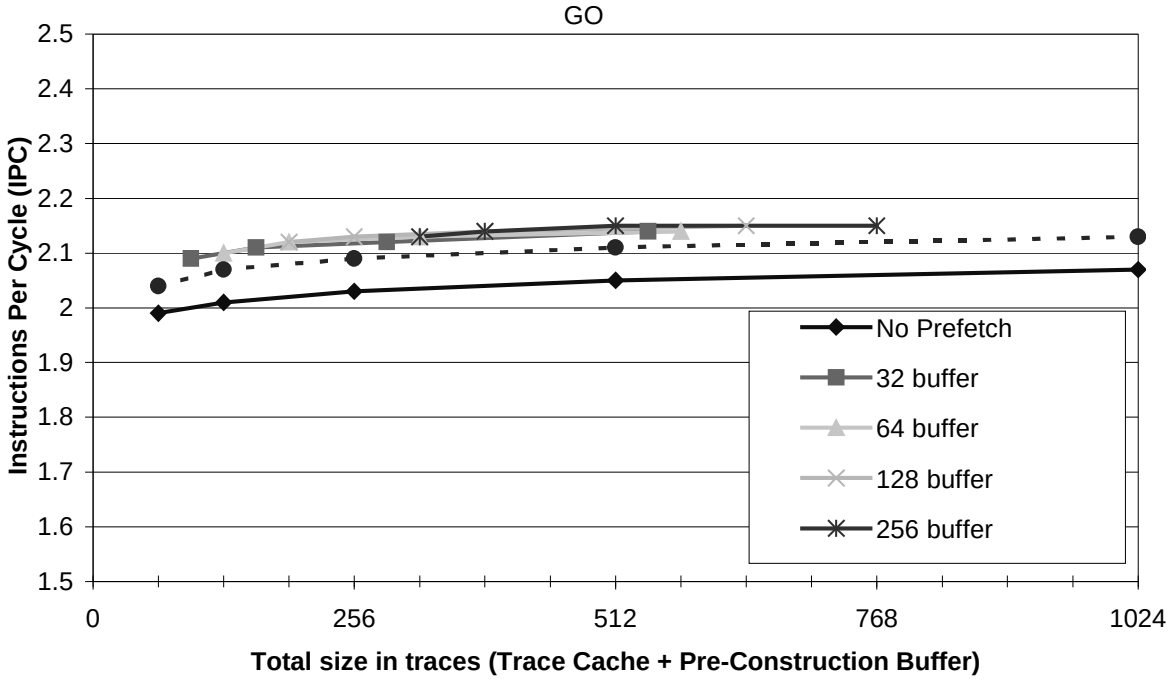


Figure 3-28 Performance for *go*.

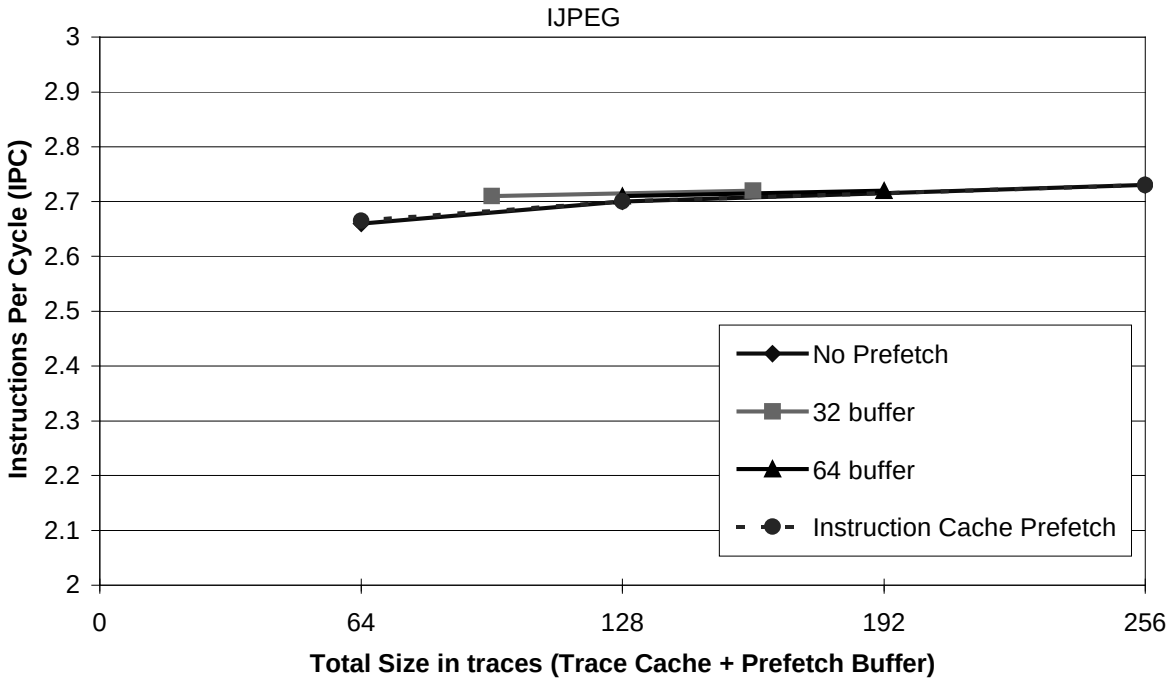


Figure 3-29 Performance for *ijpeg*.

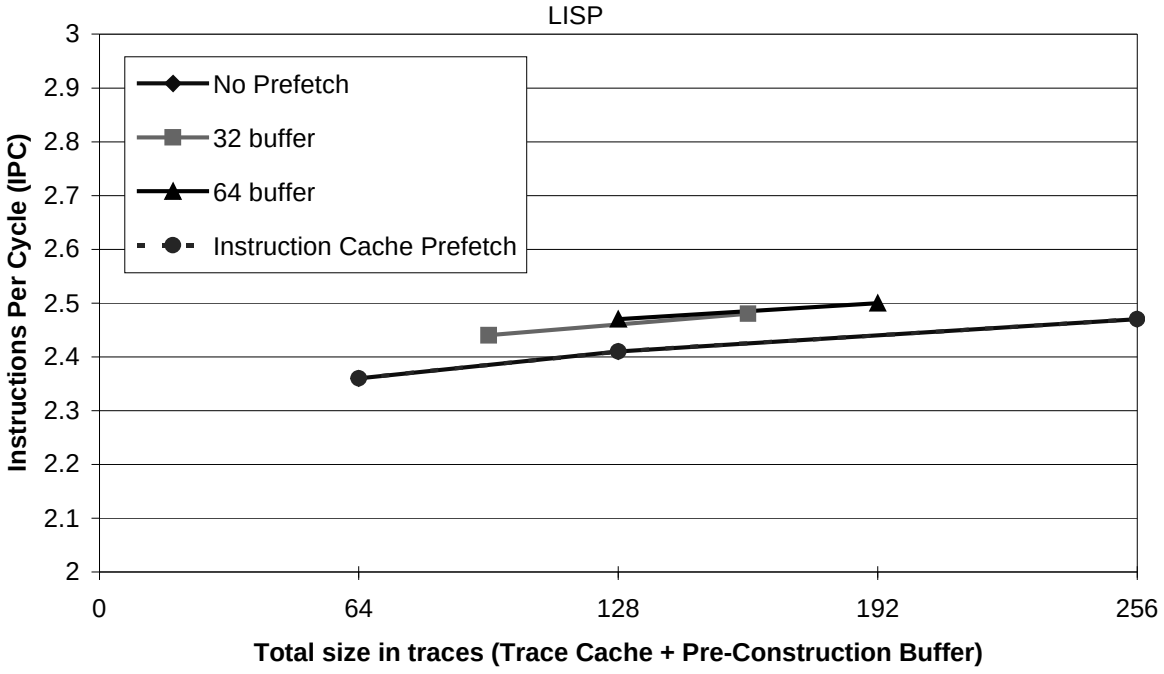


Figure 3-30 Performance for *lisp*.

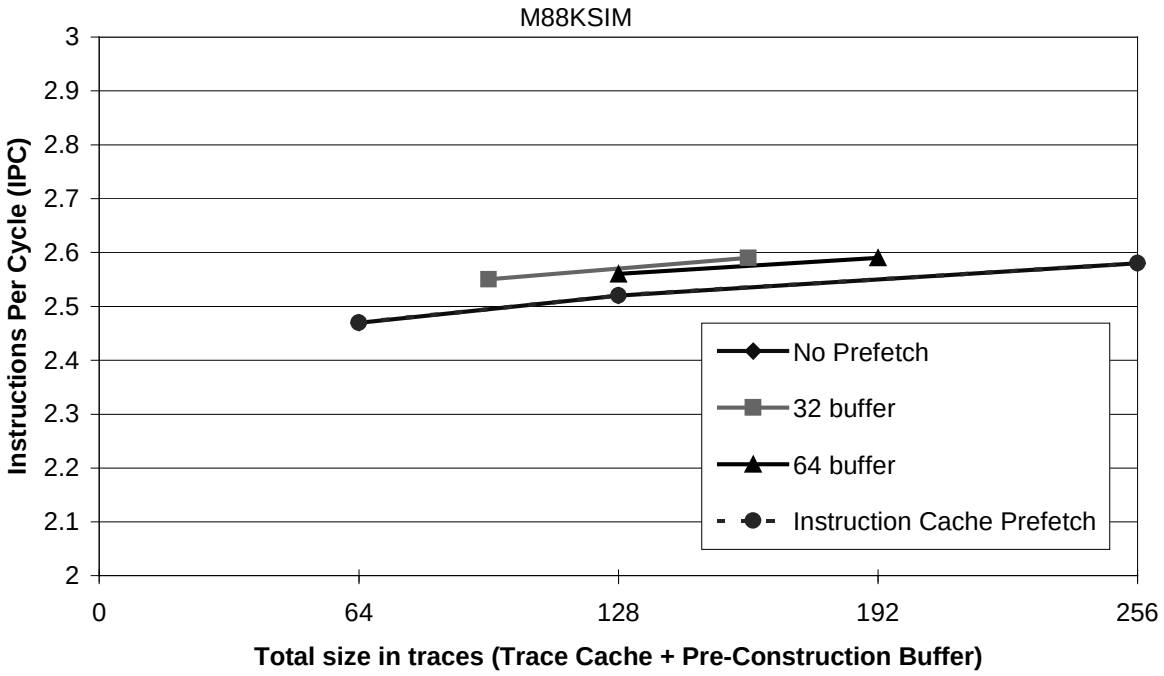


Figure 3-31 Performance for *m88ksim*.

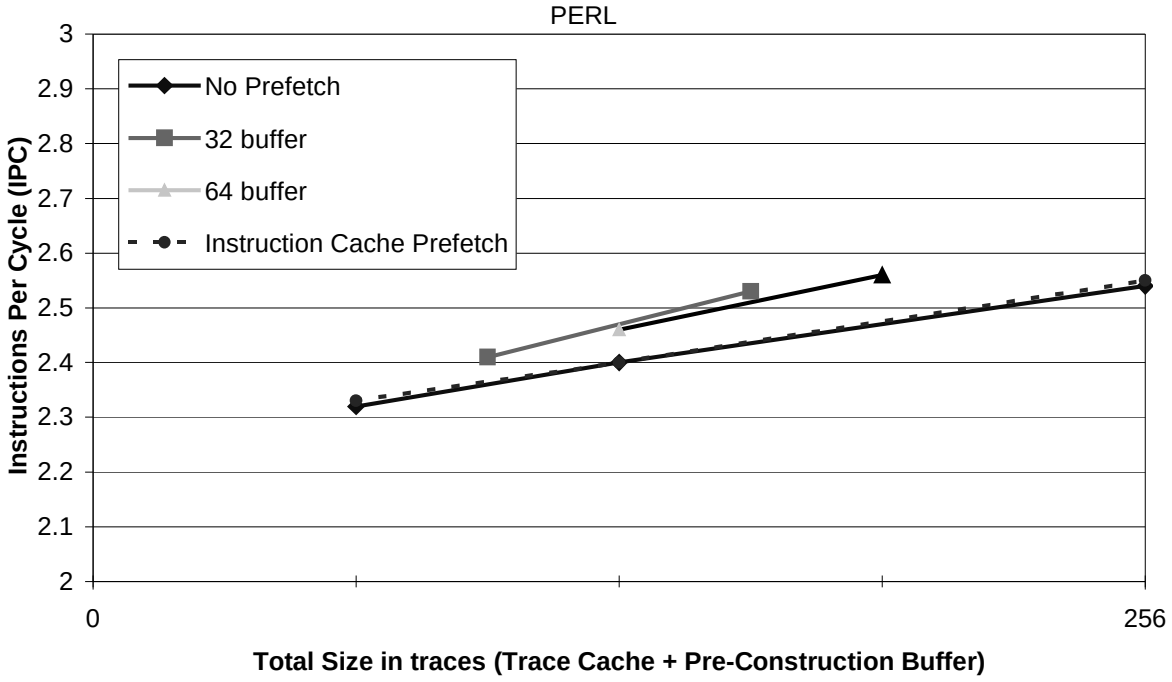


Figure 3-32 Performance for *perl*.

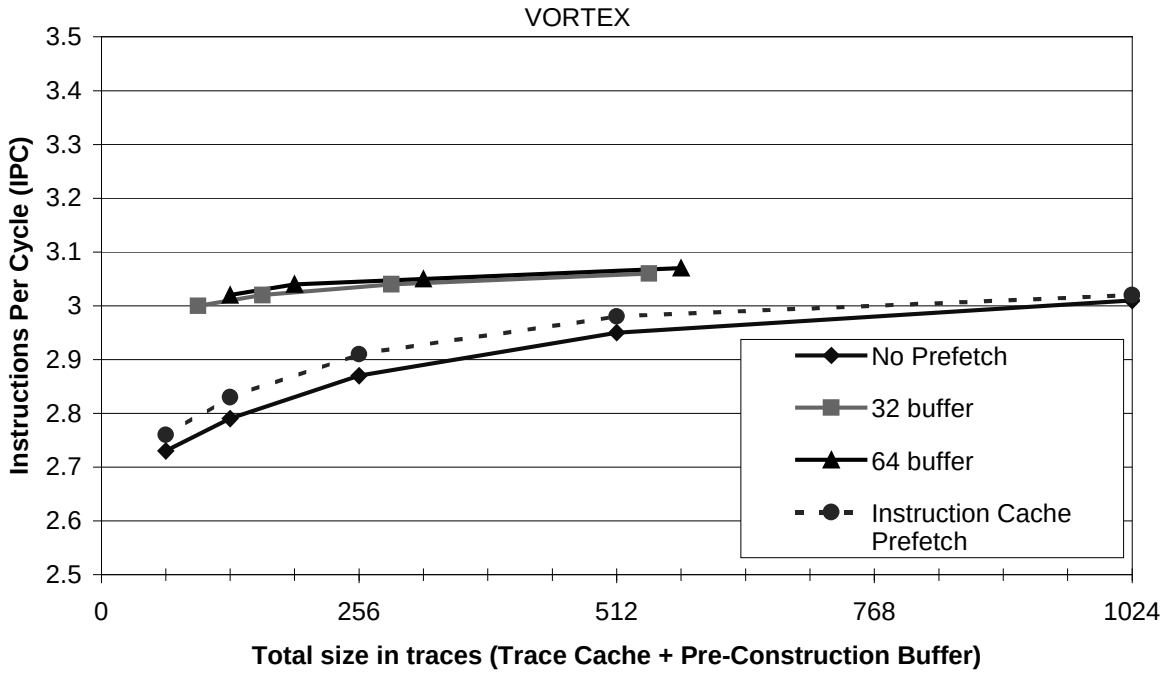


Figure 3-33 Performance for *vortex*.

3.4 Results for Transient Performance

How well processors can handle periods of transition is extremely important. Many of the mechanisms incorporated into today's processors to increase instruction level parallelism rely on dynamically learning application behavior. A straightforward example of this is the reliance of today's processors on caches and branch prediction. This reliance on dynamic information makes the performance of the processor vulnerable to periods of transition when the processor begins executing code it has not observed recently. This can occur because a context switch has brought in a different application, or because an application has reached a new region of code.

The trace cache is yet another mechanism that takes advantage of dynamically learning application behavior. Pre-construction can help make the trace cache less vulnerable to periods of transition, because it can construct traces for regions of code that have not been previously executed. This is an important aspect of pre-construction, as it increases the overall robustness of the processor.

To simulate the effects of context switches and entering new regions of code, the simulator is periodically interrupted and all the dynamic structures (caches and predictors) are flushed. This is not the best way to model real world workloads, but it does give some insight into how the processor handles periods of transition. What is important is the performance trend of the processor as the frequency of interrupts increase.

The two benchmarks *gcc* and *go* are used for these results. The first 20 million instructions are executed from the benchmarks with the processor being interrupted from

never to once every five thousand instructions. Two processor configuration are modeled, a configuration with a 512-entry trace cache, and a configuration with a 256-entry trace cache and a pre-construction engine with a 128-entry pre-construction buffer. Length 16 traces are used for all the results and the execution engine is based on the small configuration.

3.4.1 Trace cache performance during periods of transition

The first metric considered is the miss rate of the trace cache. Figure 3-34 and Figure 3-35 give the trace cache performance as a function of the frequency of interrupts for the benchmark *gcc* and *go* respectively. As the frequency of interrupts (the flushing of dynamically learned state) increases, the trace cache miss rate increases. With pre-construction, the rate of increase is less. For *gcc* the difference is small, but for *go* the difference is significant. This means that the trace cache performance is more robust with pre-construction.

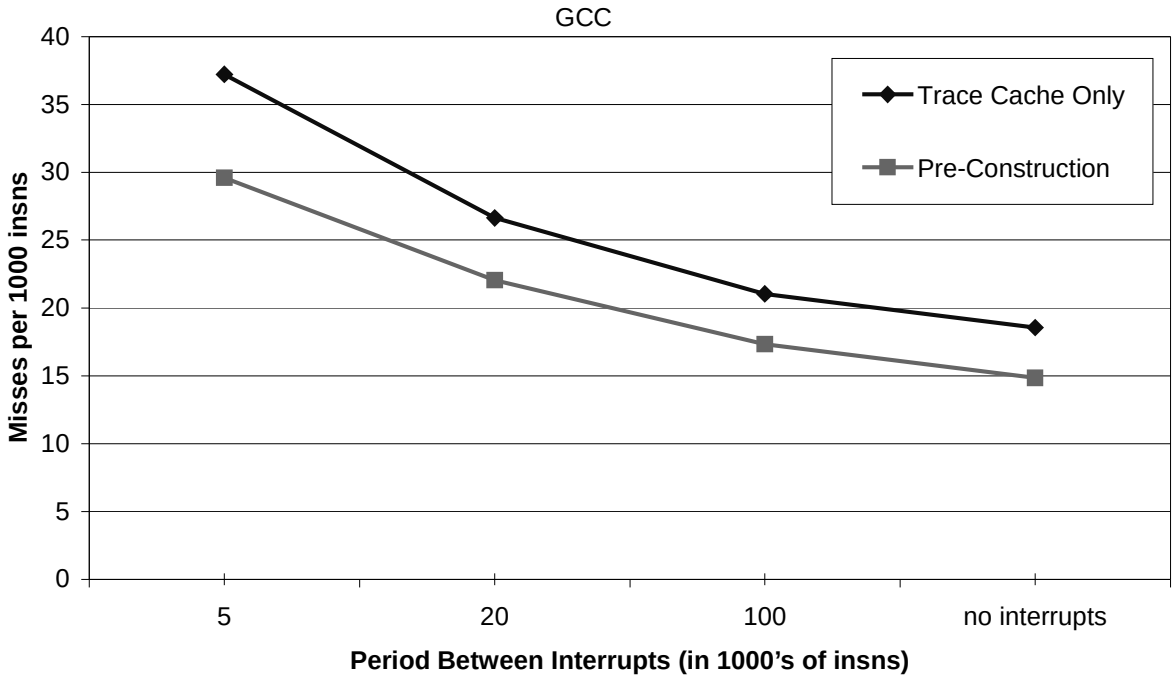


Figure 3-34 Trace cache performance in the presence of interrupts for *gcc*.

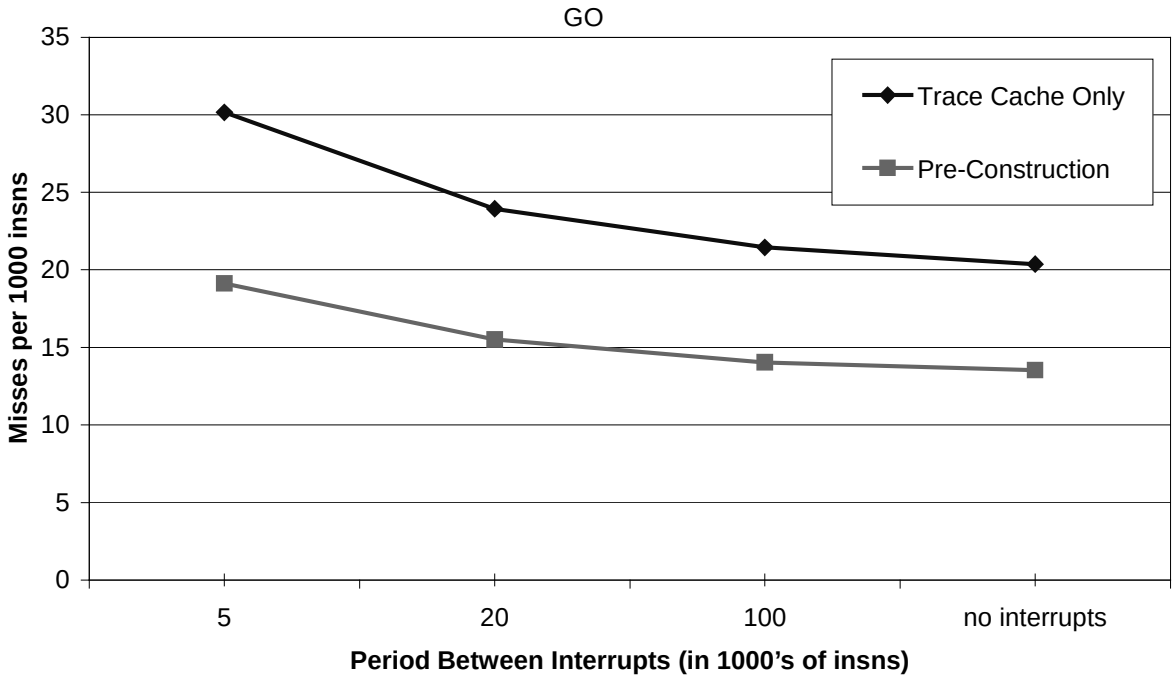


Figure 3-35 Trace cache performance in the presence of interrupts for *go*.

3.4.2 Overall performance during periods of transition

Pre-construction increases the robustness of the trace cache by making it more capable of handling periods of transition. This in turns increases the overall robustness of the processor. As the frequency of interrupts increase, the performance benefit of pre-construction also increases (see Figure 3-36 and Figure 3-37). At the extreme point of an interrupt every five thousand instructions, there is a 15% to 20% performance benefit from pre-construction for these benchmarks. That is significantly higher than the benefit observed for the same benchmarks in the steady state case.

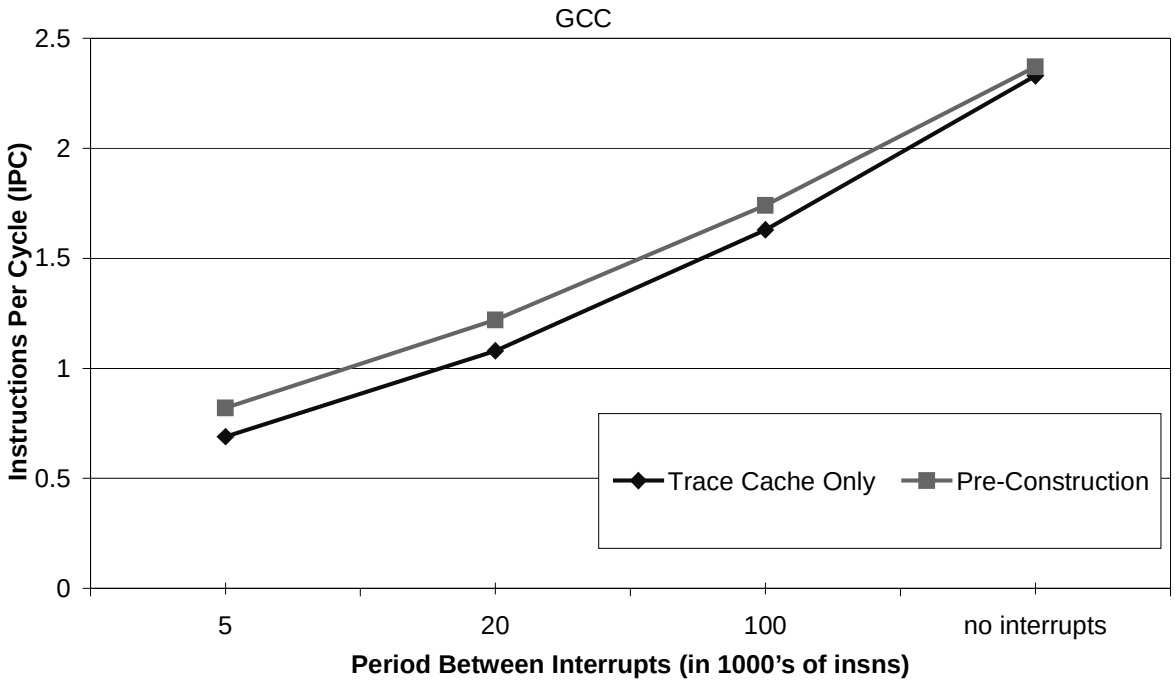


Figure 3-36 Performance of trace processor in presence of interrupts for *gcc*.

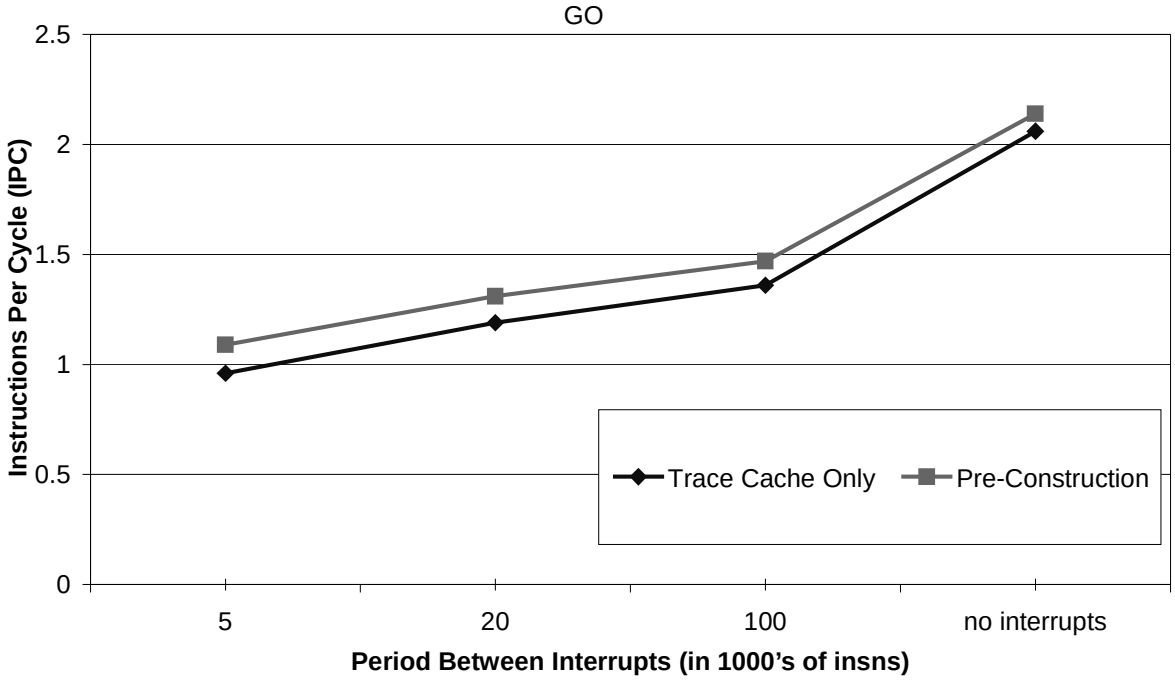


Figure 3-37 Performance of trace processor in presence of interrupts for *go*.

3.5 Results for Length 32 Traces

Pre-construction can also be applied for length 32 traces. Performance of the trace cache and pre-construction engine are similar for length 32 traces as for length 16 traces. The number of trace cache misses is less with length 32 traces for a trace cache with the same number of entries, as there are fewer unique traces for any region of code. At the same time, the cost of the trace cache, in terms of area, is twice as large for a given number of entries.

For the results based on length 32 traces, a processor with 4 processing elements each with 4-way issue is used. This configuration is used instead of the large processor configuration presented in Section 2.3 because very large instruction windows skew the trace cache performance results. A small amount of trace pre-construction is performed naturally

by a processor with a large instruction window, as potentially useful traces are constructed for processing elements following incorrect control predictions. If the control misprediction is corrected and the same trace is observed on the correct path, it appears as if the trace was pre-constructed. Doubling the instruction window from 4 processing elements each 32 instructions long to 8 processing elements each 32 instructions long reduces trace cache mispredictions by around 5% for some benchmarks. But, the reduction varies significantly between benchmarks depending on whether control paths reconverge frequently and if the reconverged paths have the same trace alignments.

Only the four benchmarks *gcc*, *go*, *perl* and *vortex* are presented. The benchmarks *lisp* and *m88ksim* see benefits similar, but slightly less than, the benchmark *perl*. The other two benchmarks have such small working sets that pre-construction does not impact performance.

3.5.1 Trace cache performance for length 32 traces

The trace cache miss rates for length 32 traces are shown in Figure 3-38 through Figure 3-41. The relative performance of pre-construction is comparable to the performance seen for length 16 traces. The actual miss rates are lower for length 32 traces, even comparing to equal size trace caches for length 16 traces. Longer traces tend to have better trace cache performance, but there are other issues that must be considered to determine the best trace length for a given processor. Transitioning between trace granularity and instruction granularity, in cases such as trace cache misses, has larger penalties for longer traces.

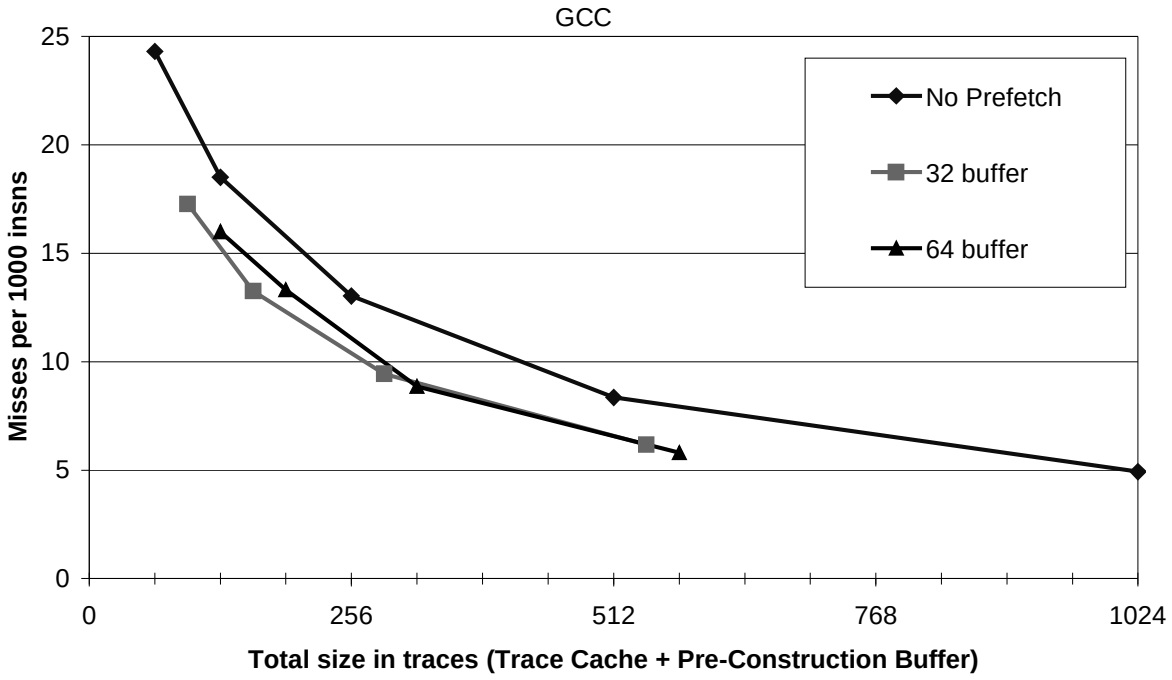


Figure 3-38 Trace cache performance for *gcc*.

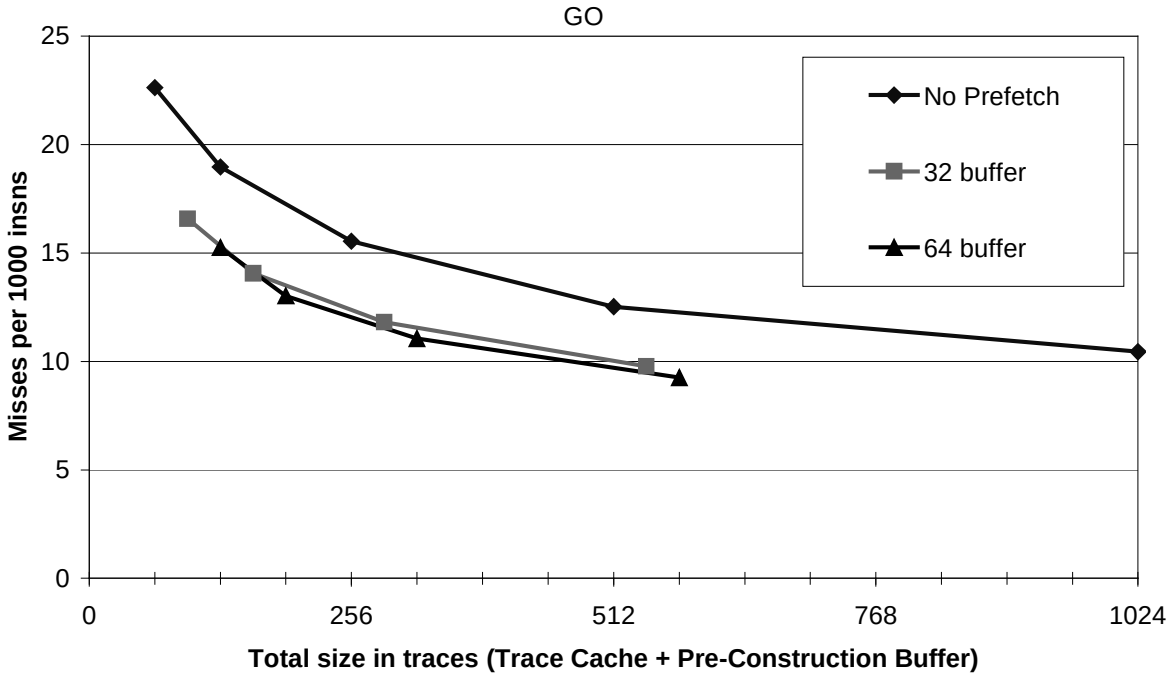


Figure 3-39 Trace cache performance for *go*.

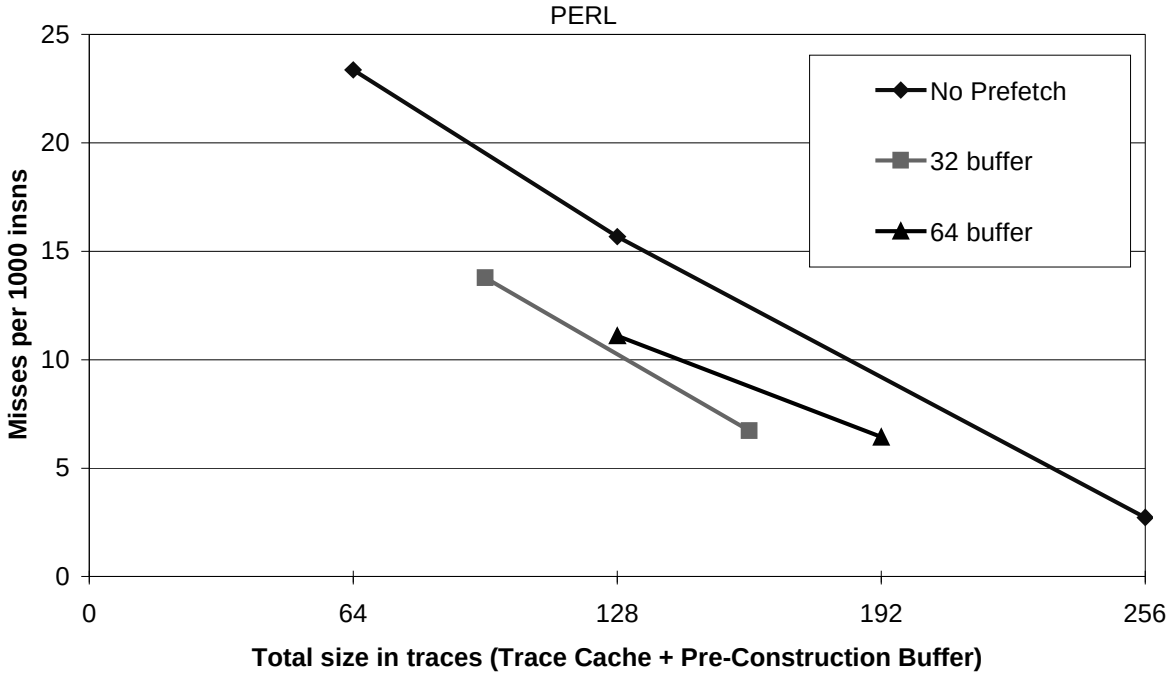


Figure 3-40 Trace cache performance for *perl*.

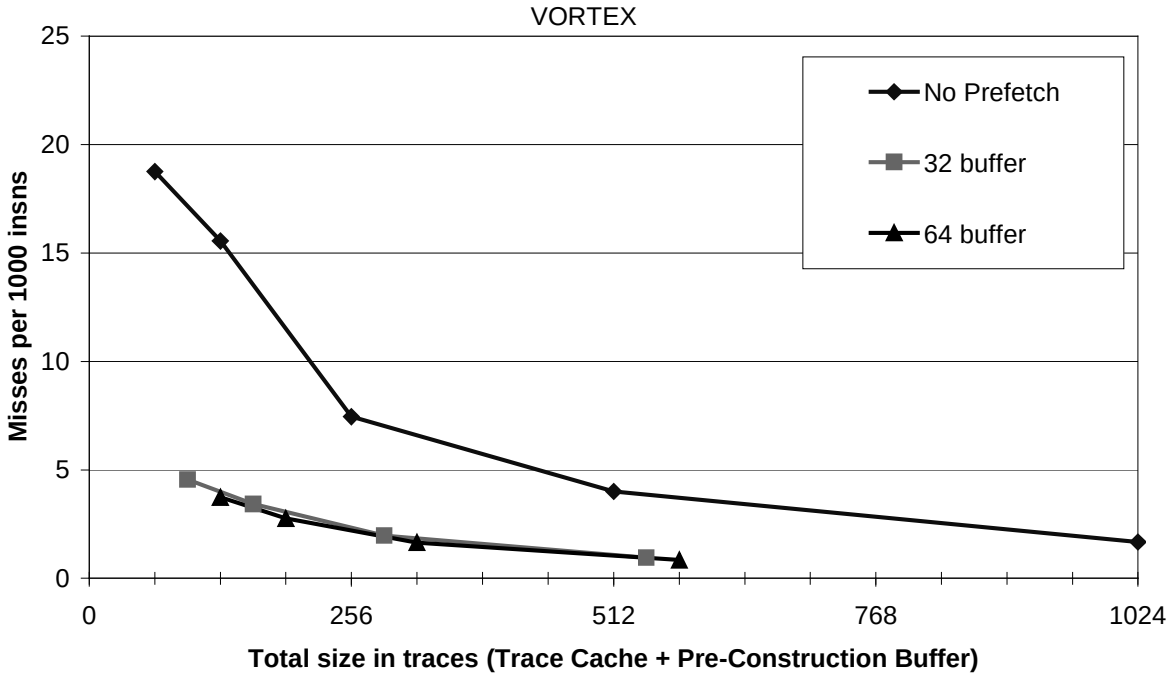


Figure 3-41 Trace cache performance for *vortex*.

3.5.2 Overall performance for length 32 traces

The overall performance results for length 32 traces are shown in Figure 3-42 through Figure 3-45. For equal size implementations, pre-construction performs from 2% to 7% better than a trace cache alone for the benchmarks shown. These results are in line with the results observed for length 16 traces.

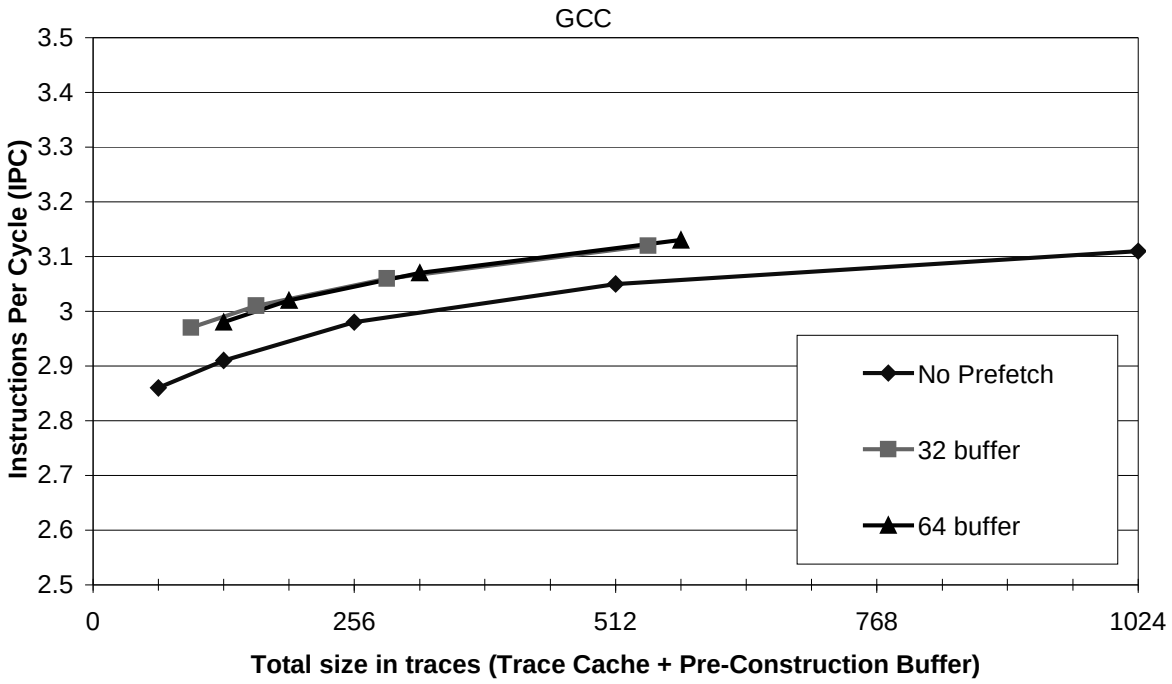


Figure 3-42 Performance of benchmark *gcc* with length 32 traces.

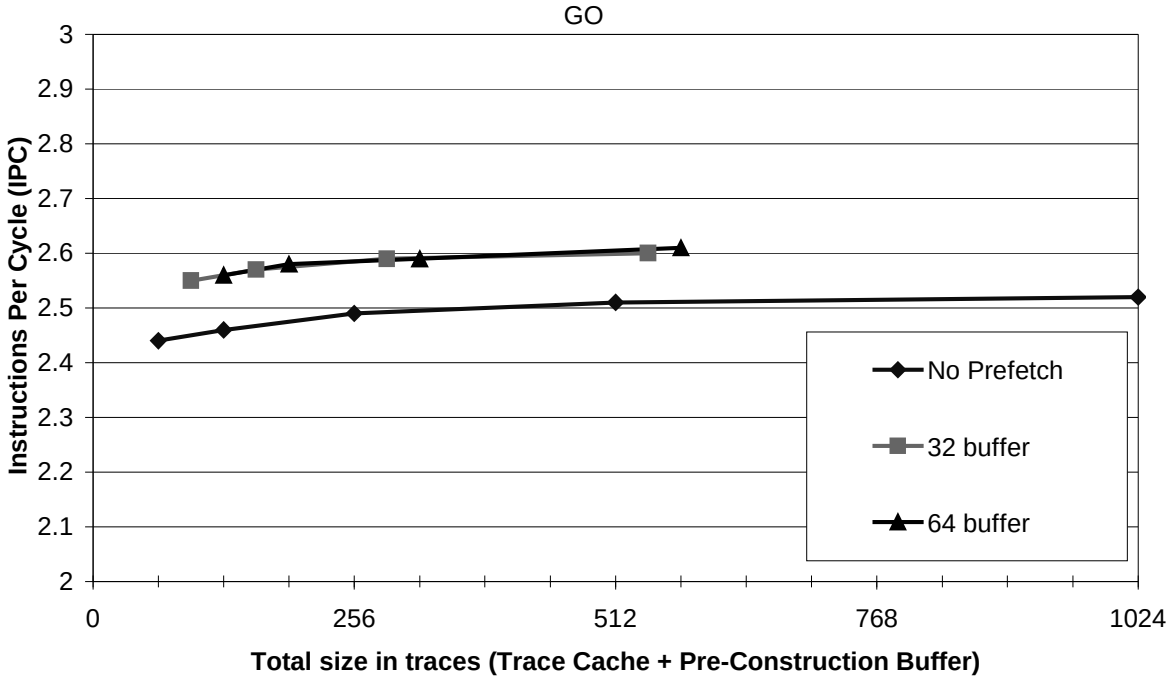


Figure 3-43 Performance for *go* with length 32 traces.

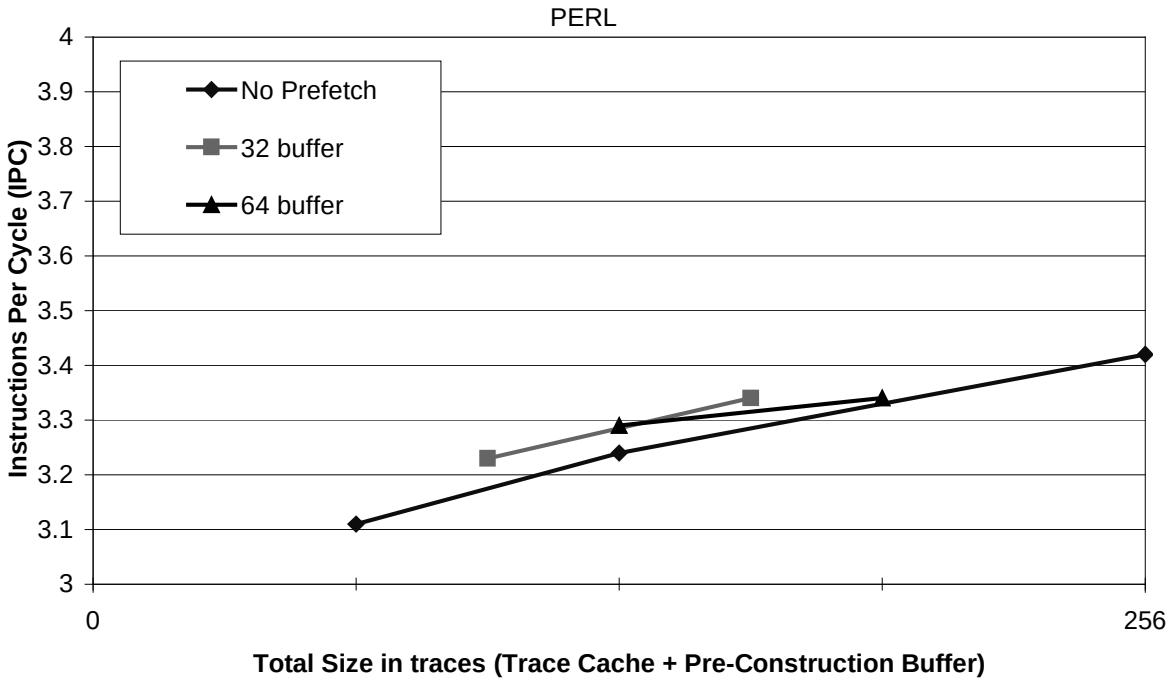


Figure 3-44 Performance for *perl* with length 32 traces.

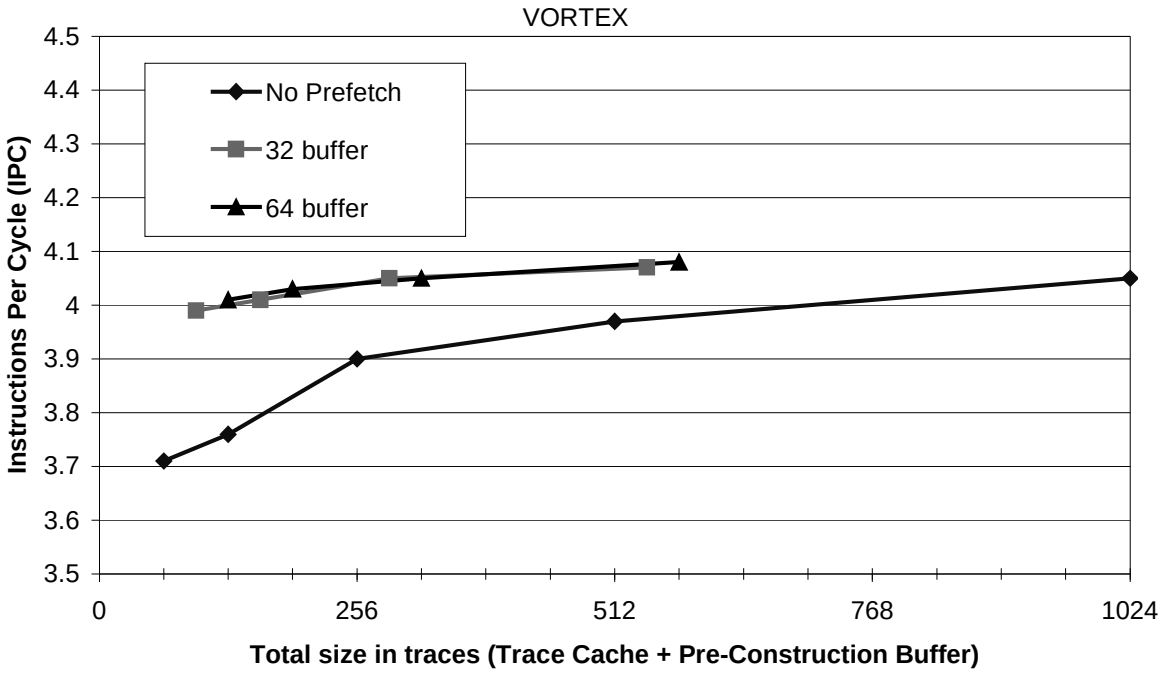


Figure 3-45 Performance for *vortex* with length 32 traces.

Chapter 4
Instruction Pre-Processing

The trace cache can enable hardware optimizations via dynamic pre-processing optimizations. Traces and the trace cache have three important roles in enabling these optimizations. First, traces serve as an intermediate representation of a program that can hold an optimized version. Second, traces represent specific dynamic paths, which enables new optimizations. Third, the trace cache can decouple the hardware that analyzes and optimizes programs from the main processor core.

The dynamic instruction stream generated from sequences of traces only needs to be functionally equivalent to the dynamic instruction stream that would otherwise be generated from fetching the static binary. This can be used to decouple an internal ISA (the instruction set the processor executes) from an external ISA (the instruction set that programs are encoded in). This is similar to the concept of using decoded instruction caches [10][11] to transform from an external ISA to an internal ISA. This could be used to implement an entirely different internal ISA from external ISA, with translation only performed in the event of a trace cache miss. Alternatively this could be used to implement an internal ISA that simply augments the external ISA with instructions to support implementation-specific optimization. I focus on the latter.

Traces can be optimized for implementation-specific hardware. The external instruction set of a processor represents an agreed-on interface between a collection of software and a family of processors. Different processors that support the same external instruction set may have significantly different internal microarchitectures. By dynamically optimizing an application for the specific hardware, significant performance benefits can be obtained.

Traces can also be used to perform optimizations that take advantage of having a specific dynamic sequence of code. Code can be specialized for a given dynamic path through a general region of code. This enables optimizations that could not be performed on the static program. By optimizing the instruction stream for implementation-specific features and specific dynamic sequences, the execution time for regions of code can be significantly reduced.

Translating to an optimized internal instruction set can be implemented in the traditional pipeline organization. The instruction decode pipeline could be lengthened to provide cycles to perform these translations and optimizations (as is done in Intel's PentiumPro processor and AMD's K6 processor). Lengthening the traditional processor pipeline has a number of drawbacks. The most significant drawback is increasing the latency to resolve mispredicted branches; this can significantly hurt performance. Lengthening the pipeline also increases the amount of speculative state that must be preserved and makes the general pipeline implementation (such as interlocking logic) more complicated. Performing these optimizations in the fetch pipeline would also require re-computing all the transformations each time a given dynamic sequence is observed.

The pre-processing pipeline organization provides an ideal environment for implementing the translation to the optimized instruction set. If the processor core can execute unoptimized instructions, then the translation hardware can be decoupled from the processor core. The pre-processing pipeline can perform the translation and optimization of instructions and replace entries in the trace cache with optimized entries. If the processor core can only execute optimized instructions, than transformation has to be performed on instructions generated by the slow path fetch logic before the instructions are fed to the execution engine. In this case, the translation only needs to be performed in the event of trace cache misses.

4.1 Overview of Instruction Pre-Processing

Instruction pre-processing takes advantage of traces and the trace cache to provide an optimized instruction stream to the processor backend. I study instruction pre-processing in the context of a trace processor with a distributed execution engine. Most, if not all, the optimizations discussed for pre-processing would be equally applicable to a processor that uses a trace cache for instruction supply but uses a traditional homogenous superscalar execution engine.

4.1.1 Beyond compiler optimizations

Pre-processing is intended as a complement, not as a substitute, for good optimizing compilers. The dynamic nature of traces and implementation dependence allows for additional transformations that would be impossible, or at least very difficult, at compile

time. There are three inherent characteristics of traces that enable pre-processing optimizations.

First, traces represent a specific dynamic path through a region of code. Traces can potentially contain instructions from multiple basic blocks; and a given instruction can appear in multiple traces. Each trace sequence places its instructions within a specific context, and optimizations applied across basic block boundaries only have to be valid for that specific context. This enables specializing code for specific dynamic paths through regions of code. Compilers have attempted to specialize code for specific dynamic paths with some limited success [13].

Second, the encapsulation of traces as units of work can make the usage and availability of resources (both execution and communication resources) more predictable. In the proposed trace processor microarchitecture, each trace is executed on its own processing element. This makes the availability of the specific hardware resources known at the time a trace is optimized. In addition, inter-trace data value communication, i.e., external values consumed and produced by a trace, is determined as the trace is formed, and the trace can be optimized for this communication.

Finally, traces provide an intermediate representation of programs that can enable application binaries, which may be compiled for a large family of processors, to be dynamically tuned for a specific processor core. Pre-processing can target new, internal instructions that the processing elements can execute, but which are not supported in the

external instruction set. These internal instructions can provide performance features in a way that is transparent to the original binary.

4.1.2 Range of optimizations

The most important part of this work is creating the framework for optimizations and identifying the important information available for optimizations. The range of optimizations that could be incorporated into the pre-processing framework is almost boundless. I look at three specific optimizations. These optimizations illustrate the variety of optimizations that are possible and demonstrate how the characteristics of traces can be exploited during pre-processing. The three optimizations are:

- **Scheduling instructions.** This optimization is traditionally done in software by compilers and in hardware by out-of-order instruction issue. By scheduling within traces, pre-processing hardware can take advantage of more sophisticated scheduling algorithms than is done by the highly-localized out-of-order issue. By having the dynamic context available, pre-processing hardware can provide a more custom-tailored schedule than can be done at the static level by the compiler. Furthermore, the information regarding use and availability of resources can be used by the scheduling heuristic to make the best use of the processor hardware.

- **Performing constant propagation.** This optimization is traditionally done by compilers, but including it as part of instruction pre-processing allows optimizations beyond basic block boundaries that only have to be correct within the context of the specific control flow implied by the trace.

- **Collapsing data dependent chains of computation to single instructions**. This optimization uses compound instructions which may have a lower total latency than the dependence chain of individual instructions. For example, a three-input ALU can be built with essentially the same latency as a two-input ALU [63]. Although three-input compound instructions may not be part of the machine's standard instruction set, these can be implemented in the hardware for use after instruction pre-processing.

4.2 Incorporating Instruction Pre-Processing

A high-level view of how pre-processing is incorporated into the processor is shown in Figure 4-1. When the next trace of a program is not in the cache, the slow path hardware constructs a trace, using branch prediction as needed. This newly constructed trace is dispatched to an idle processing element to be executed and the new trace is also placed in the trace cache (in non-optimized form). In parallel with the initial execution of the trace most of the work of pre-processing is performed. When the trace is completed (given that it was a correctly predicted trace) the pre-processor performs the final transformations on the trace. We assume a two-cycle latency for this transformation. The new optimized trace is then inserted into the trace cache. Using this model, the latency of instruction pre-processing is not on the critical path.

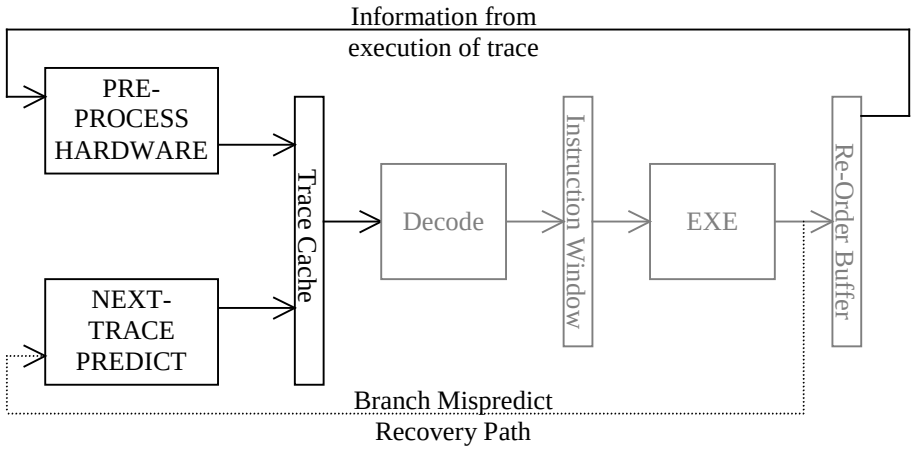


Figure 4-1 Incorporating instruction pre-processing

4.3 Specific Optimizations

4.3.1 Instruction scheduling

The first optimization considered is instruction scheduling. As a practical matter, instructions are not literally moved within the trace. Such movement would probably consume time or add to hardware complexity; furthermore, the original instruction order must be "remembered" so that precise traps can be correctly implemented. Rather, we prioritize the instructions within a trace and rely on out-of-order issue logic to complete the scheduling task by following these pre-established priorities. That is, when there are more instructions ready to issue simultaneously than there are resources, the instructions marked with higher priority will be issued first.

Instruction scheduling attempts to order instruction execution to achieve the lowest execution time with the available execution resources. An upper bound on what instruction scheduling can achieve is the performance of a processor with an issue bandwidth equal to

the window size (no competition for execution resources). The more constrained the execution resources, the more opportunity instruction scheduling has to increase performance.

Determining the ideal schedule for instruction execution is a hard problem, and when only a portion of the instruction window is available (one trace) at a time it is impossible to determine the global optimum. Not only is there the matter of determining the critical path based on data dependencies, there are issues of resolving and recovering from incorrect control predictions and accommodating variable length, and potentially long length, memory operations.

I do not investigate ideal scheduling, but instead look at some simple heuristics that have the potential to increase performance. There are many possible heuristics that can be used, either alone or in combination, to assign scheduling priorities within a trace. Adjusting the priority of an instruction usually requires related adjustments in other instructions belonging to the same dependence chain. However, because the same instruction may belong to multiple dependence chains, there may be conflicts among different scheduling heuristics; i.e., one heuristic may attempt to give an instruction a higher priority, while another may attempt to give a lower priority. The priority and interactions of the heuristics become very important.

I investigate three scheduling heuristics, alone and in conjunction. The heuristics take advantage of simple program characteristics that make intuitive sense. The heuristics are:

- **Scheduling for the early resolution of branches.** When a branch is mispredicted, the latency of branch verification is critical to performance. For all branches, or for those branches that have been identified as frequently mispredicted, it is important to schedule the instructions so that the branch outcome is computed as early as possible. Initially all branches are given high priority. After a trace has been executed, only those branches that have been observed to be mispredicted more than 12.5% of the time are given high priority and other branches are given lower priority.

- **Scheduling for early execution of memory operations.** Memory operations should be scheduled early as they are often high latency operations.

- **Scheduling for inter-trace communication.** Propagating values across task boundaries requires an extra cycle of latency because of the distributed nature of the execution resources [49][58]. It is important to schedule instructions to minimize the impact of this extra latency. Instructions that produce values needed by subsequent traces should be scheduled as early as possible. And instructions that consume values generated by previous traces should be scheduled as late as possible to allow for the extra latency. Often a chain of computation will begin by consuming a value generated by an earlier task and end with the production of a value used by later tasks, creating a conflict of interest. In this case the instructions are given high priority to schedule them early. The policy of scheduling for inter-trace communication is similar to the work in the multiscalar compiler to schedule for inter-task communication [64].

Each heuristic is initially explored independently. Then the heuristics are combined, giving precedence to the heuristics with the most benefit. This is not intended to be an exhaustive search of the design space of instruction scheduling. This work is intended to show the type of optimizations that are possible and to demonstrate that benefits can be seen from simple heuristics.

4.3.2 Constant propagation

The second pre-processing optimization considered is constant propagation within traces. There are many cases where computation chains with no inputs are used to generate constant values. These can occur because of control dependences that affect the value being produced or because of encoding limitations in the instruction set. With traces both of these restrictions can be overcome.

Control dependences restrict the range of constant propagation in static programs. As noted earlier, a trace represents a specific control path through a region of a program. There may be constant values that can be propagated for a specific control path that can not be propagated in the static program due to control dependences. A good example of traces overcoming control dependences occurs when a trace contains the initialization and the first few iterations of a simple *for* loop. In this case, the value initially set for the loop count can be propagated through the first few increments/decrements.

Simplescalar (the external instruction set), like most RISC instruction sets, only supports 16-bit immediate operands for most operations. When word length immediate operands are required, two instructions are necessary (a load-upper immediate and an OR/ADD

immediate) to load the immediate value into a register so that other instructions can use the value. In the internal instruction set, word length immediate fields can be supported; this transformation increases the space required to hold the instructions but can significantly reduce the number of instructions that need to be executed. A common case where this optimization can be beneficial is generating global addresses for loads/stores. Because of the cost of supporting word length operands, we consider both implementations that do and do not support this feature.

Constant propagation leads to performance improvement in two ways. First, there are some instructions that do not need to be executed after constant propagation. If there are limited execution resources this removes contention and can speed execution of a trace. If the instructions removed by constant propagation put a value into a register that is a live-out value, the value still needs to be propagated to subsequent traces. In this case a directive to forward an immediate value replaces the instruction. These directives do not require execution resources but do compete for global result bus bandwidth. Instructions removed by constant propagation do not need to generate local register values, as any local instruction will have the register dependence replaced by an immediate operand.

The second way that constant propagation increases performance is by reducing the length of dependence chains. For example, if in the original program there is a *load upper immediate* followed by a *logical OR immediate* followed by an *add* operation this is a chain of three dependent instructions that has to execute in series. With constant propagation the three instructions can be replaced by an *add* instruction with a word length immediate

operand. If the add instruction's other operand is available, the transformed instructions can operate with two fewer cycles of latency.

4.3.3 Instruction collapsing

There has been much discussion recently of the impact of the relative increase in wire delays to gate delay [29][52]. The flip side of the relative increase in wire delays is the relative decrease in gate delays. While much current research is motivated by the problems posed by long wire delays there has been little work performed to investigate the opportunities posed by short gate delays. Developing more sophisticated execution resources can significantly reduce the latency of executing a program. With the relative decrease in the delays of gates, adding a few levels of logic to execution resources may not impact the cycle time.

Most instruction sets only support the encoding of operations having two source operands. We explore the possibility of incorporating more complex operations into the internal instruction set of the processor and then transforming the instructions within a trace to take advantage of these operations. More complex operations can be used to collapse small chains of dependent operations into a single operation, thus reducing the latency. In general, data collapsing turns a chain of dependent operations into a single operation with more inputs (see Figure 4-2). This can lead to increased performance if the new operations do not increase the base cycle time of the processor. With the relative decrease of the delay of gates with respect to wires, these new operations are likely feasible.

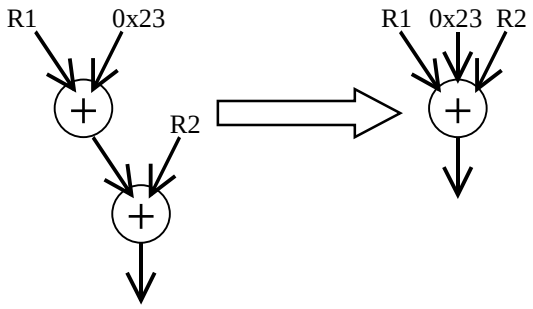


Figure 4-2 Example of instruction collapsing.

Instruction collapsing is implemented by replacing the last instruction in a computation chain with a new instruction that captures the functionality of all the instructions in the chain. Other instructions may use the results created by the earlier instructions in the computation chain, however. If other instructions need the results created by earlier instructions, then the earlier instructions must be kept and executed. In this case, there is no reduction in the number of instructions executed, but the latency to produce the result of the collapsed instruction is reduced.

On the other hand, after collapsing is performed, some instructions no longer need to be executed. An instruction can only be removed if no instruction within the trace uses its result and no instruction beyond the current trace can use the result (the result is not a live-out value). We investigate the performance impact of either executing or not executing these instructions. Not executing the instruction removes unnecessary work and reduces contention for execution resources. Executing the instruction simplifies the maintaining of precise state for recovering from control mispredictions or exceptions. If a misprediction or exception occurs in the middle of a trace, the later instructions of the trace will be discarded and potentially new instructions will be brought in to complete the trace. In this event, the data dependences change, and instructions that were optimized away may need to be

executed. Executing all instructions also simplifies the logic for performing collapsing as it does not require the determination of which instructions can be safely removed.

We will study a range of data collapsing configurations to discover where and how it can best be utilized. Data collapsing can be used to collapse chains of either heterogeneous or homogeneous operations, and there are an almost limitless number of computational chains that can be replaced by specialized hardware to form a single operation. We only consider collapsing of fixed-point operations; similar optimizations could be applied to the floating-point units. There are a few common operations that can capture most of the benefit of data collapsing.

Most of the benefits of collapsing can be obtained by adding a new operation that only has two register source operands and single register destination operand (see Section 4.5.3). This is very important as this operation does not change the demands on the register file and data bypassing configurations, both of which are communication intensive and are usually timing critical. In addition to the two register operands, the new operation incorporates three immediate source operands and can collapse a number of common chains of dependent operations. The new operation was determined by observing what computation chains were collapsed by the aggressive collapsing policy described in Section 4.5.3.

The new operation consists of shifting each of the register source operands left by a small immediate amount (0 to 3), negating either or both operands, and adding the two values and a third immediate value to produce the result (see Figure 4-3). This operation is extremely powerful but only adds a couple of gate delays over the traditional two operand

add instruction. The two largest portions, in terms of time, of a traditional add instruction is the communication latency (including the bypass delays) and the carry propagate logic. The immediate shift adds a 4-to-1 mux and using a three-operand addition only adds a carry save adder (the latency of a single-bit adder).

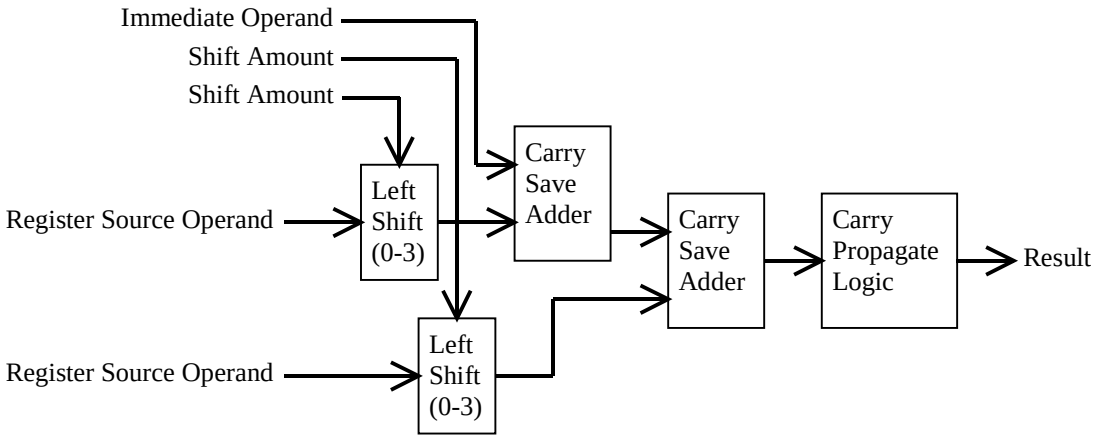


Figure 4-3 New arithmetic unit.

Whether the additional latency will affect the cycle time of a processor can only be answered by a full processor design project. Such a procedure is well beyond the scope of this work; however, the additional latency is minimal and would likely not affect the cycle time of a processor. Given the assumption that this operation can be used, I will show that it can lead to significant performance improvements.

4.4 Implementing Pre-Processing

Much of the work of instruction pre-processing is overlapped with the first execution of a trace. Final transformation is performed on the trace after it is executed, as described in Section 4.2. A two-cycle latency is modeled for this transformation. The details of implementing each of the optimizations are discussed in the following sections.

4.4.1 Instruction scheduling

As mentioned above, when instructions are scheduled, the instructions are not literally reordered, instead a small priority field is added to each instruction. An increase in priority needs to be propagated to all other instructions on which the instruction depends. Dependence analysis is straightforward because local registers are renamed based on instruction position in the trace. The hardware to perform the scheduling could likely be incorporated into the processing elements' instruction windows, so the work could be performed while the trace is being executed.

In this implementation a straightforward approach to scheduling is used. A complete ordering of instructions (by priority) is maintained. This requires a field associated with each instruction that is of size log base 2 of the number of instructions in the trace (4 or 5 bits). Initially all instructions are ordered by their position in the trace. Priorities are adjusted to give priorities according to the heuristics used, while keeping the restriction that an instruction must have a lower priority than an instruction that it is dependent on.

In a real implementation, a simpler approach would likely be taken (i.e., coarser granularity of priority). A reasonable implementation would likely add a couple of bits to each instruction that could be used to mark them as higher priority. These fields would simply be incremented for certain classes of instructions, as well as all instruction on which they depend. This field would then be incorporated in the issue logic to determine which instruction to issue when multiple instructions were ready.

The instruction scheduling optimization does not effect program correctness or precise interrupt support. Scheduling simply provides hints for the out-of-order execution logic. This minimizes the cost of implementing the optimizations, both in terms of hardware and engineering time.

4.4.2 Constant propagation

Constant propagation can be incorporated into the initial execution of a trace. This is performed by adding a bit to the local register file indicating whether a value was generated from only constant operands. Instructions propagate the bit to their destination if all their sources have the bit set. Instructions producing a constant value record the value, and they are marked to inhibit their execution in the future. Register source operands with the constant bit set are replaced by the constant value for future executions. More simply stated, traces record values that will never change so that they do not need to be recomputed each time the trace executes.

As mentioned in Section 4.3.2, two types of constant propagation are considered. The first only considers cases where a constant value fits in the constant field of the external ISA, 16 bits for most instructions. In this case the optimization could likely be implemented without adding any extra state to the trace. An instruction needs to be able to indicate that its result value is encoded in the constant field and that the instruction does not need to execute, this could likely be done with an unused opcode. An instruction also needs to indicate that a source operand is not needed and that a constant is used instead, this could likely be done by translating to corresponding instructions that have immediate operands.

The second type of constant propagation makes use of larger constant fields than are available in the external ISA. This case will require larger instruction encodings. If the external ISA supports 16-bit constant fields and the internal ISA supports word length constant fields, the internal encoding would have to be that much larger (16 bits larger for a 32-bit machine). This optimization would have a significant impact on the size of the trace cache by making each entry larger.

The constant propagation optimization can be implemented while providing precise interrupts. At the trace boundary the precise state is available, so if a whole trace is to be retired there is no side effect of having not executed the instructions removed by constant propagation. If a branch misprediction or trap occurs in the middle of the trace, the instructions before the event, removed by constant propagation, must write their results to the local register file (this does not require execution, just result bandwidth).

4.4.3 Dependence collapsing

Dependence collapsing is implemented in two phases. The first phase is implemented during the initial execution of the trace. A small type field is added to the local register file specifying the type of the instruction that creates the register value. Dependent instructions detect whether they can be collapsed with producing instruction(s). In this way the candidates for collapsing are determined. The second phase is taking dependent chains of instructions and forming the new compound instruction that captures the same functionality but has lower total latency. The compound instruction replaces the last instruction in the chain.

The compound instructions will require additional encoding space. Exactly how much extra space is required depends on which combinations of instructions are allowed to be collapsed. For incorporating the one new compound instruction, a couple extra bits of opcode space and approximately 9 bits of extra operand information would likely be required. The extra space for operand information is needed to hold short immediate values to specify how much to shift operands, as well as an extra register source operand.

For the two types of collapsing investigated, there is a significant difference in how control mispredictions and interrupts are handled. In one, instructions that are no longer needed after collapsing are still executed. In this case, there are no problems in recovering from control mispredictions and implementing precise interrupts. Transformed instructions simply use the values generated by earlier instructions directly. If control mispredictions or exceptions occur, the correct architected state is always available. In the case that the unneeded instructions are not executed, recovering from control mispredictions and implementing precise interrupts is more complicated. At trace boundaries the correct architected state is available, but at internal points of the trace some architected state may have been optimized away. To support correct behavior, the unneeded instruction(s) must be executed if a misprediction or exception occurs in the middle of a trace. This can be implemented by adding a bit to mark these instructions to inhibit their execution, but if a misprediction or exception occurs the bit is cleared and the instructions must be executed.

4.5 Performance

The three processor configurations from Section 2.3.4 are considered. A *small* configurations with length 16 traces, 4 processing elements and 2-way issue per processing element (64 entry instruction window and 8-way issue total). A *medium* configuration with length 16 traces, 8 processing elements and 2-way issue per processing element (128 entry instruction window and 16-way issue total). And, a *large* configuration with length 32 traces, 8 processing elements and 4-way issue per processing element (256 entry instruction window and 32-way issue total).

The processor configurations modeled all have a 512 entry, 2-way set-associative trace cache. For the models with length 16 traces, the trace cache holds 32KB worth of instructions. For the model with length 32 traces, the trace cache holds 64KB worth of instructions. Pre-construction is initially not considered. At the end of the chapter, Section 4.6 discusses the integration of pre-construction and pre-processing.

Not all traces dispatched to processing elements for execution have been optimized. When a trace is dispatched to a processing element there are three possible cases: the trace was just constructed on the slow path, the trace was in the trace cache in non-optimized form, or the trace was in the trace cache in optimized form. The fraction of traces from each of these cases is shown in Table 4-1 (these results are based on the medium configuration and the combined scheduling heuristic, there is some variation based on configuration and exact optimizations performed). These results are for traces that retire, and do not include traces for incorrect paths. Between 63% and 100% of traces dispatched are optimized. Of the

traces that are not optimized, most are due to the trace not being in the cache. Only the benchmarks *gcc* and *go* see a significant number (5% and 11% respectively) of traces provided by the trace cache in unoptimized form. This is a large enough number to justify placing the unoptimized trace in the trace cache until the optimized one become available, otherwise these cases would become trace cache misses.

Table 4-1 Breakdown of traces dispatched to processing elements.

Benchmark	% of traces provide by slow path	% of traces provided by trace cache (non-optimized)	% of traces provided by trace cache (optimized)
compress	0	0	100
gcc	20	5	74
go	26	11	63
ijpeg	1	0	99
li	2	1	97
m88ksim	3	0	96
perl	8	2	90
vortex	14	1	85

In the next sections the three main pre-processing optimizations are examined. First, each optimization is considered in isolation. Then the optimizations are studied in combination to see the full potential of instruction pre-processing. The SPECint95 benchmarks are used for the results presented in this section. Corresponding results for the SPECfp95 benchmarks are presented in Appendix A.

4.5.1 Instruction scheduling

We identified three scheduling heuristics in section 4.3.1. The scheduling heuristics were based on branches, memory operations, and inter-trace communication. Each scheduling heuristic is tried alone, then in combination. The speedups from incorporating

instruction scheduling are shown in Figure 4-4 through Figure 4-11. The first bar of the graphs provide upper bounds for instruction scheduling, the next three bars show the speedup from each of the heuristics in isolation and the last bar shows the speedup from combining all the heuristics.

The first bar of the graphs shows the performance of a processor with issue width equal to the window size for both memory and non-memory operations, this is an upper bound on what scheduling could possibly achieve. The upper bound ranges from a 3% to 18% speedup across the different benchmarks and processor configurations (with an average of 12%).

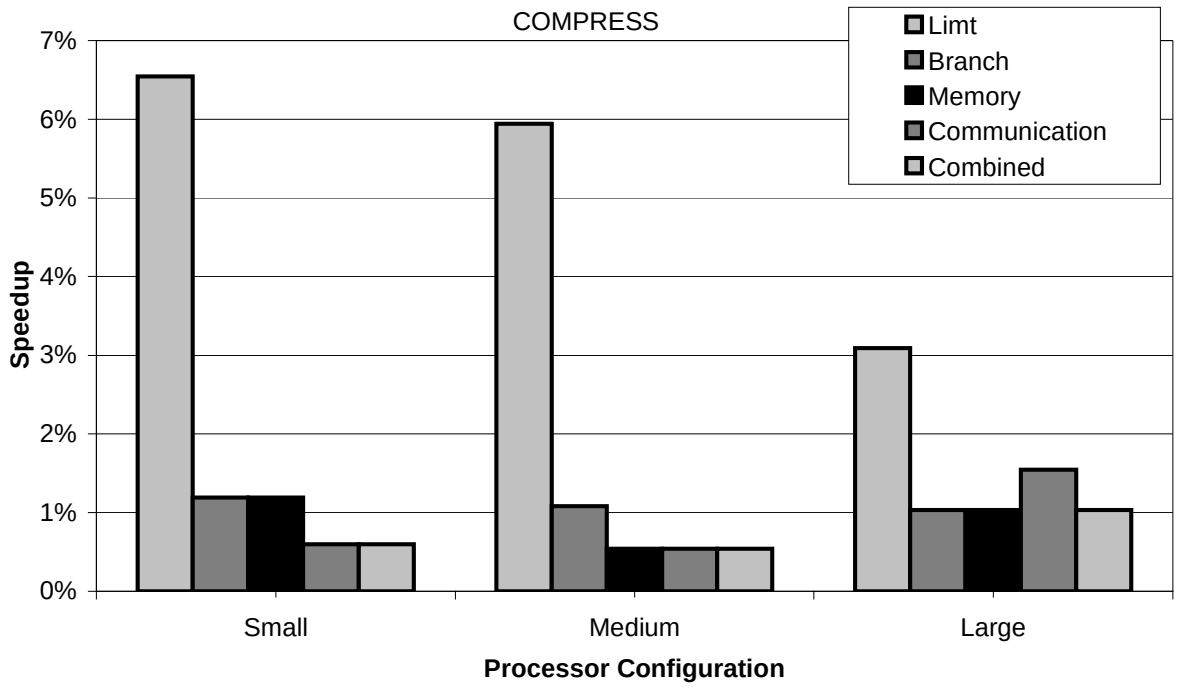


Figure 4-4 Speedup from scheduling optimization for *compress*.

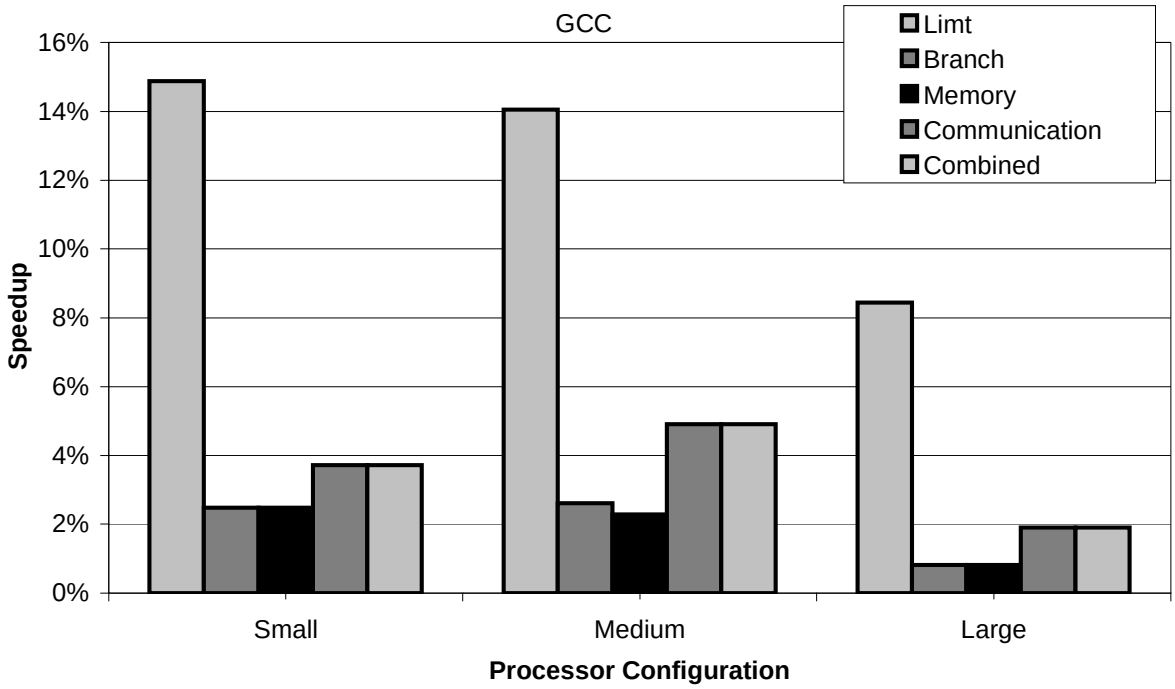


Figure 4-5 Speedup from scheduling optimization for *gcc*.

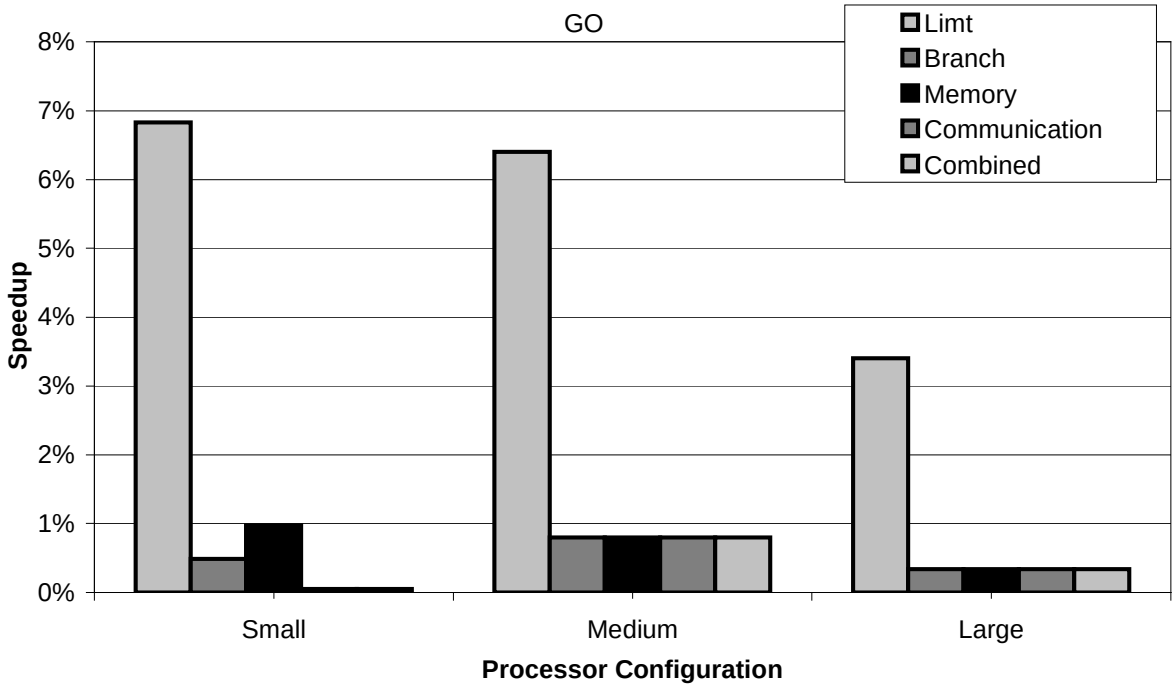


Figure 4-6 Speedup from scheduling optimization for *go*.

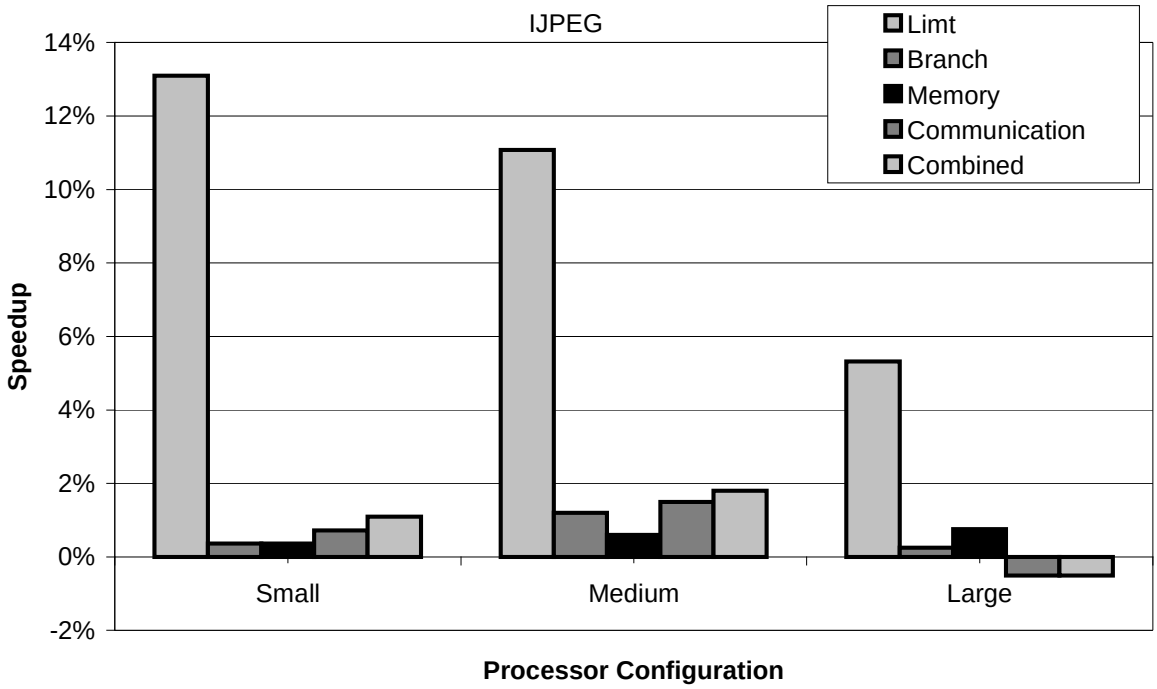


Figure 4-7 Speedup from scheduling optimization for *ijpeg*.

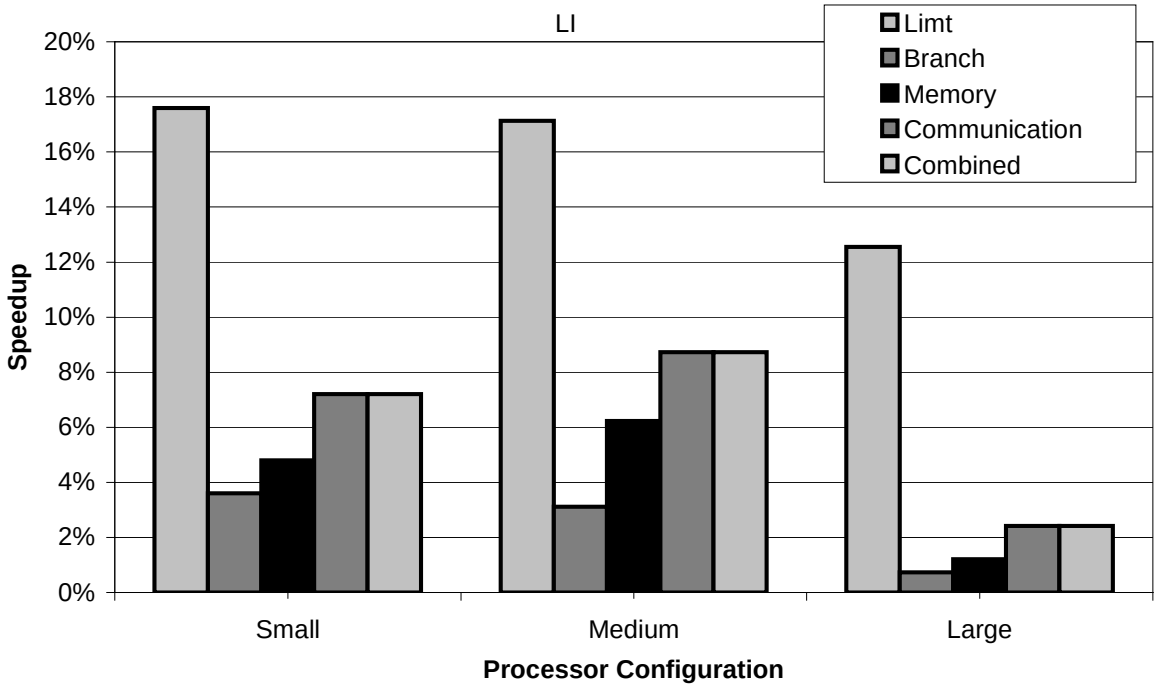


Figure 4-8 Speedup from scheduling optimization for *li*.

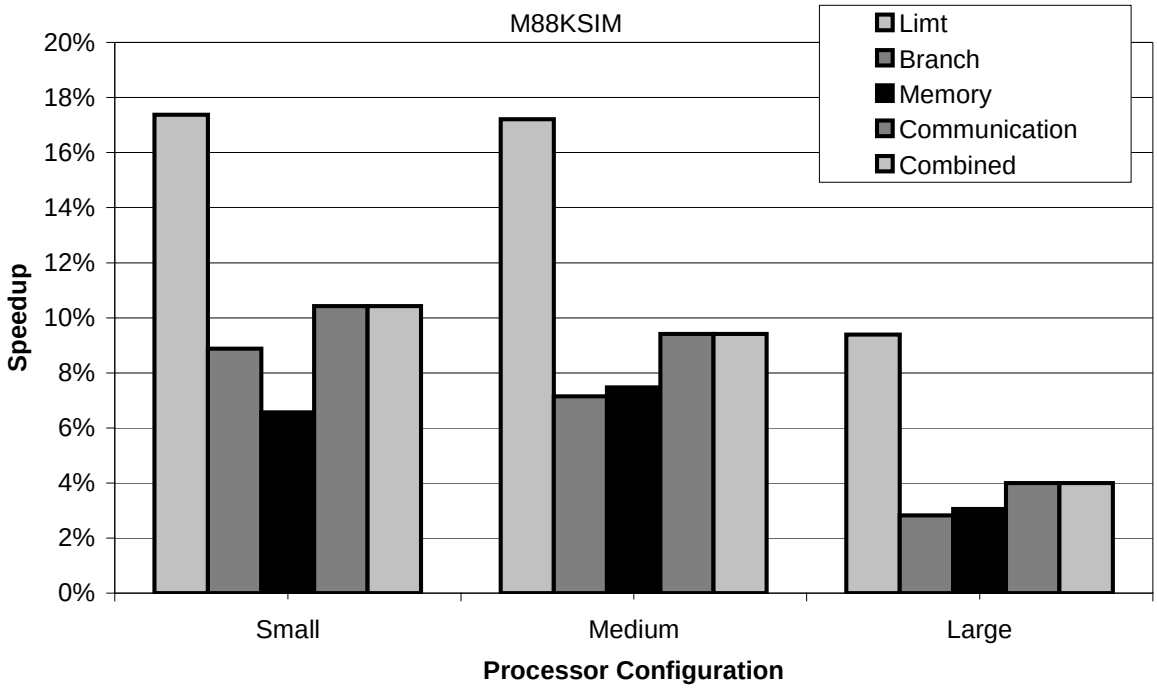


Figure 4-9 Speedup from scheduling optimization for *m88ksim*.

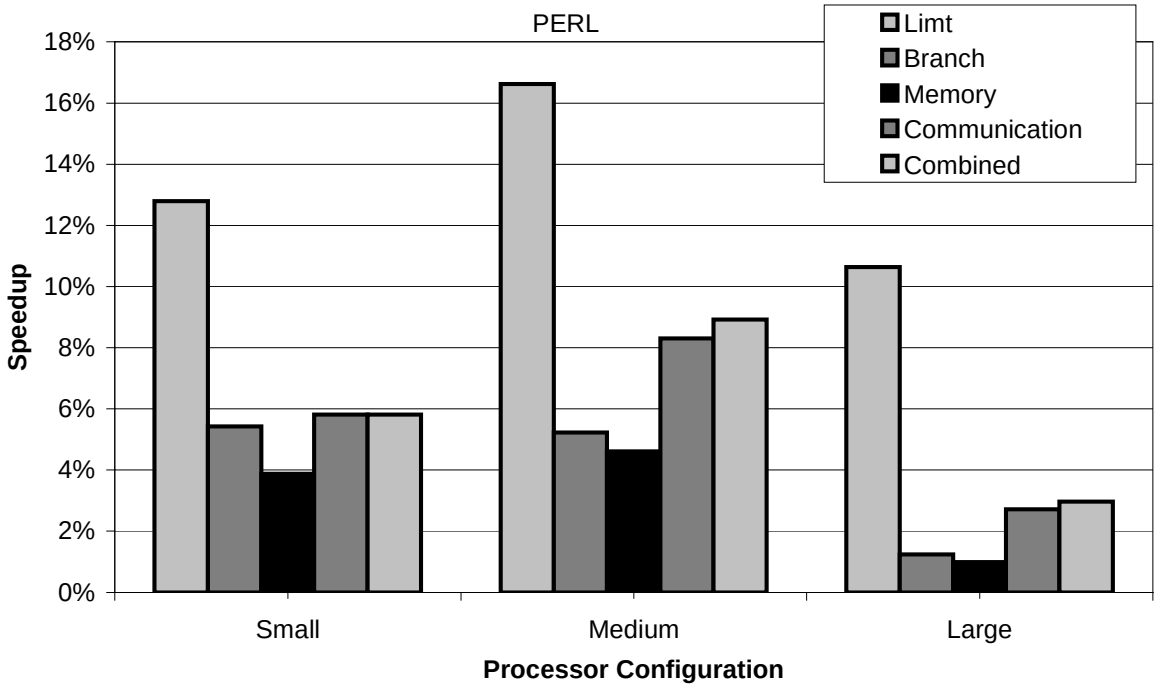


Figure 4-10 Speedup from scheduling optimization for *perl*.

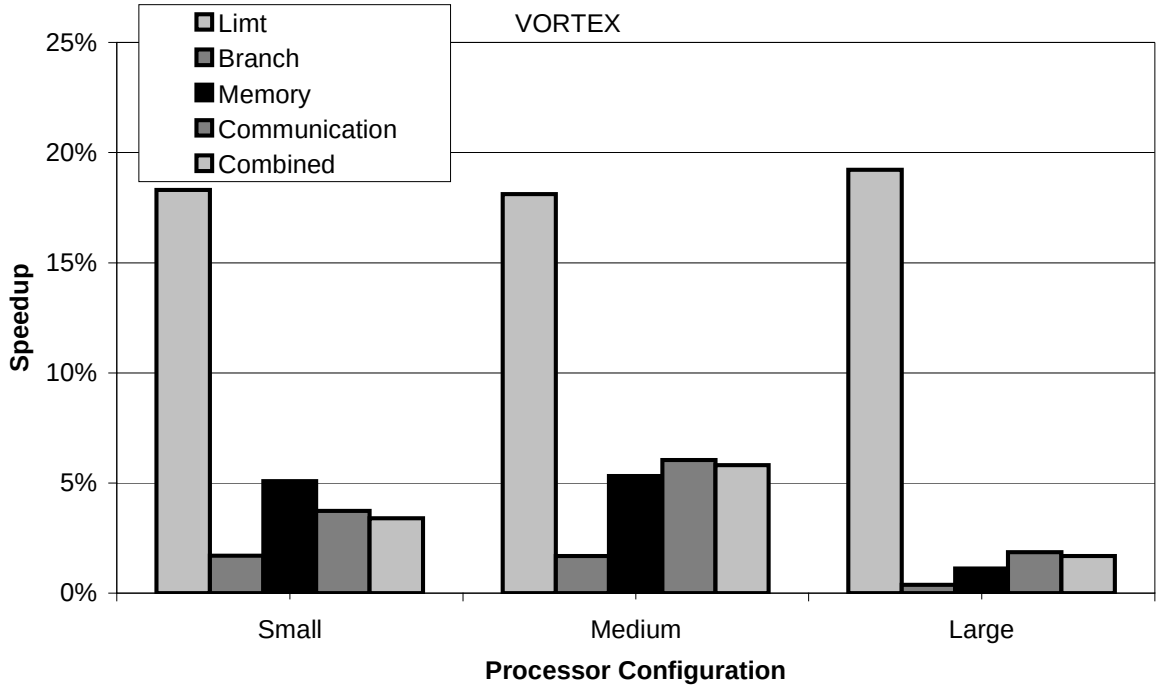


Figure 4-11 Speedup from scheduling optimization for *vortex*.

All the benchmarks see a speedup from the scheduling heuristic for branches and the scheduling heuristic for memory operations. The branch heuristic produces a 0% to 9% speedup, but the harmonic mean of the speedup is only 1%. The speedup for the memory heuristic performs comparably, performing better than the branch heuristic for some benchmarks and worse for others. The harmonic mean of the speedup from the memory heuristic is also only 1%.

Of the three heuristics, scheduling for communication has the largest benefit. The inter-trace register communication is critical to performance and, because traces are unknown at compile time, the compiler can not help schedule for this. Scheduling for inter-trace communication can produce nearly a 10% speedup in some cases, but in an equal number of cases the speedup is negligible. In the case of the benchmark *ijpeg* there is a slight slowdown

from the scheduling heuristic. The average speedup from scheduling for communication is 4%, 5% and just under 2% for the small, medium and large processor configurations respectively.

The combined scheduling approach uses all three policies, but not with equal priority. Scheduling for communication is given the most influence on the scheduling, then scheduling for memory operations, and lastly scheduling for branches. The combined scheduling is performed by iteratively rescheduling for each heuristic in increasing order of priority.

In general the combined approach provides the greatest benefit, but the benefit over scheduling for communication-only is minimal. Of the eight SPECint95 benchmarks, three (*li*, *m88ksim* and *perl*) see between a 5% and 10% speedup for the small and medium configurations, two (*gcc* and *vortex*) see smaller speedups and three (*compress*, *go* and *ijpeg*) see no significant speedups. The average speedup of the combined scheduling heuristic is 4% and 5% for the small and medium configurations respectively (harmonic mean of 3% and 2%).

The large processor configuration sees minimal benefit from scheduling (average speedup of less than 2% for the combined scheduling heuristic). Scheduling is only useful in cases when there are more instructions ready to issue then there is issue bandwidth available. By doubling the issue bandwidth per processing element the probability of this happening decreases, even though the number of instruction competing for the bandwidth is also doubled. In general, the more flexibly the issue bandwidth is divided among the instructions, the less likely that there will be ready instructions that can not issue. This suggests that there

would be minimal benefit from instruction scheduling in a large homogenous superscalar execution engine.

When other optimizations are applied to increase instruction level parallelism, such as instruction collapsing, the benefit of scheduling increases. By increasing the number of instructions that can issue in parallel, the probability that the number of ready instructions will exceed the issue bandwidth increases. This is shown in the section on combining instruction pre-processing optimizations (Section 4.5.4).

The performance benefit of scheduling comes from rearranging the priorities of instructions within traces. Two interesting measurements are how much are instructions rearranged and how many instructions are moved past control transfer instructions. A breakdown of how far instructions are moved during scheduling is shown Figure 4-12. These results are based on the scheduling for communication heuristic and the medium processor configuration. Figure 4-13 shows how often instructions are moved past at least one control instructions during scheduling.

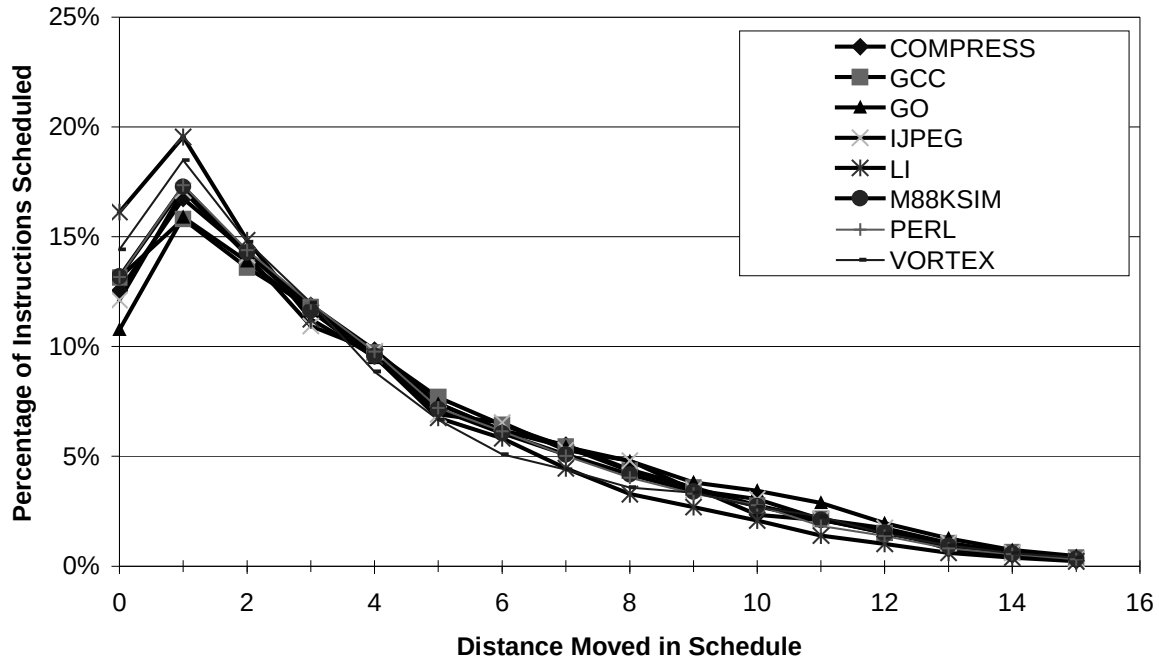


Figure 4-12 Average distance instructions are moved during scheduling.

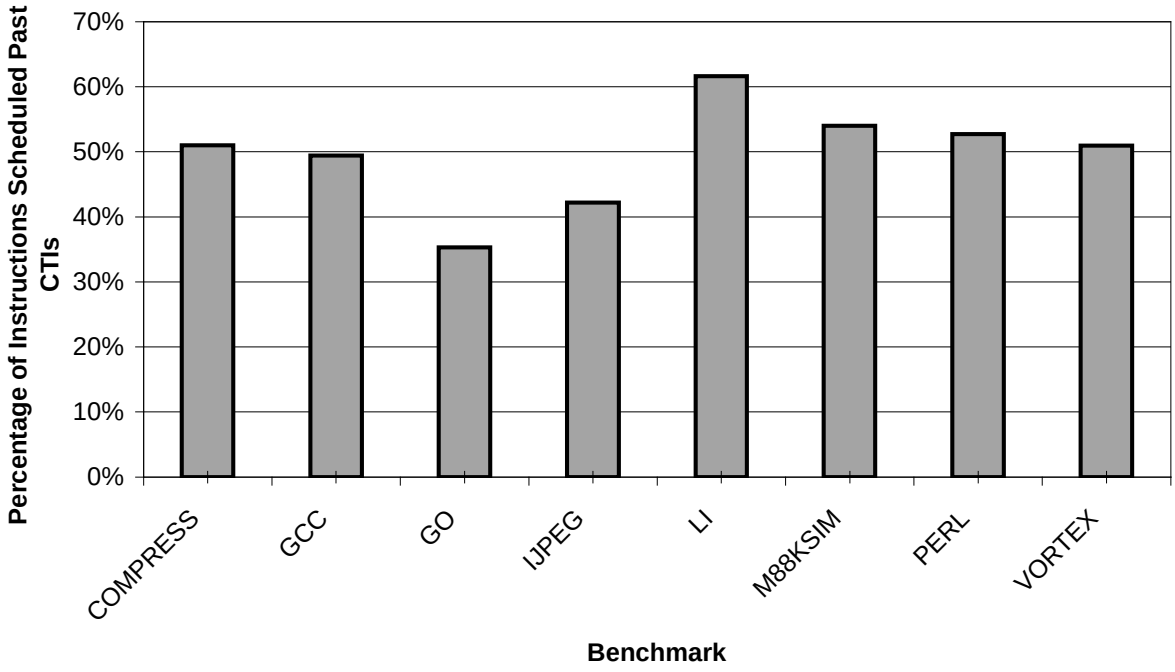


Figure 4-13 Percentage of instructions moved past control instruction during scheduling.

4.5.2 Constant propagation

Constant propagation removes computations that do not need to be recomputed each time a trace is reused. It is limited in that it can only propagate constants within a trace but has the advantage of being based on a specific dynamic path and being able to use a more flexible internal instruction encoding.

Table 4-2 and Table 4-3 show the percentage of instructions removed by constant propagation for length 16 and length 32 traces respectively. As discussed in Section 4.3.2, two implementations of constant propagation are considered. The difference between them is whether instructions can encode word length constant operands.

Table 4-2 Percentage of instructions removed by constant propagation for length 16 traces.

Benchmark	% of instructions removed by constant propagation with limited size constant operands	% of instructions removed by constant propagation with word size constant operands
compress	1.7	4.9
gcc	4.8	7.3
go	2.6	12
ijpeg	3.5	4.2
lisp	3.5	4.0
m88ksim	3.8	9.6
perl	2.9	5.2
vortex	5.9	6.6

Table 4-3 Percentage of instructions removed by constant propagation for length 32 traces.

Benchmark	% of instructions removed by constant propagation with limited size constant operands	% of instructions removed by constant propagation with word size constant operands
compress	2.9	6.6
gcc	5.5	8.2
go	2.7	12
ijpeg	4.9	5.7
lisp	3.7	4.2
m88ksim	4.3	10
perl	4.0	6.6
vortex	7.7	8.5

A significant number of instructions can be removed by constant propagation. This reduces the number of instructions competing for issue bandwidth and also reduces the length of computation chains in the program. When word length constant operands are allowed, from 4% to 12% of instructions can be removed by constant propagation for length 16 traces. Slightly more can be removed for length 32 traces. Without changing the instruction encoding, and only taking advantage of the single dynamic path within a trace, from 3% to 6% of instructions can be removed for length 16 traces. For length 32 traces the number goes up to 3% to 8%. Given the limited scope of this optimization, this is an impressive result.

Although a significant fraction of instructions are removed in some cases by constant propagation, the speedup is minimal (see Figure 4-14 and Figure 4-15). With the exception of the large processor configuration of *m88ksim* (3% speedup) all the benchmarks see less than a 2% speedup from constant propagation. The reason for the minimal speedup is that with the large instruction window and accurate prediction, computation based only on constants can usually be computed before the values are needed. This is especially true in the

trace processor configuration where execution resources are dedicated to traces. Constant propagation would probably have a larger impact if small pools of execution resources were shared by multiple processing elements.

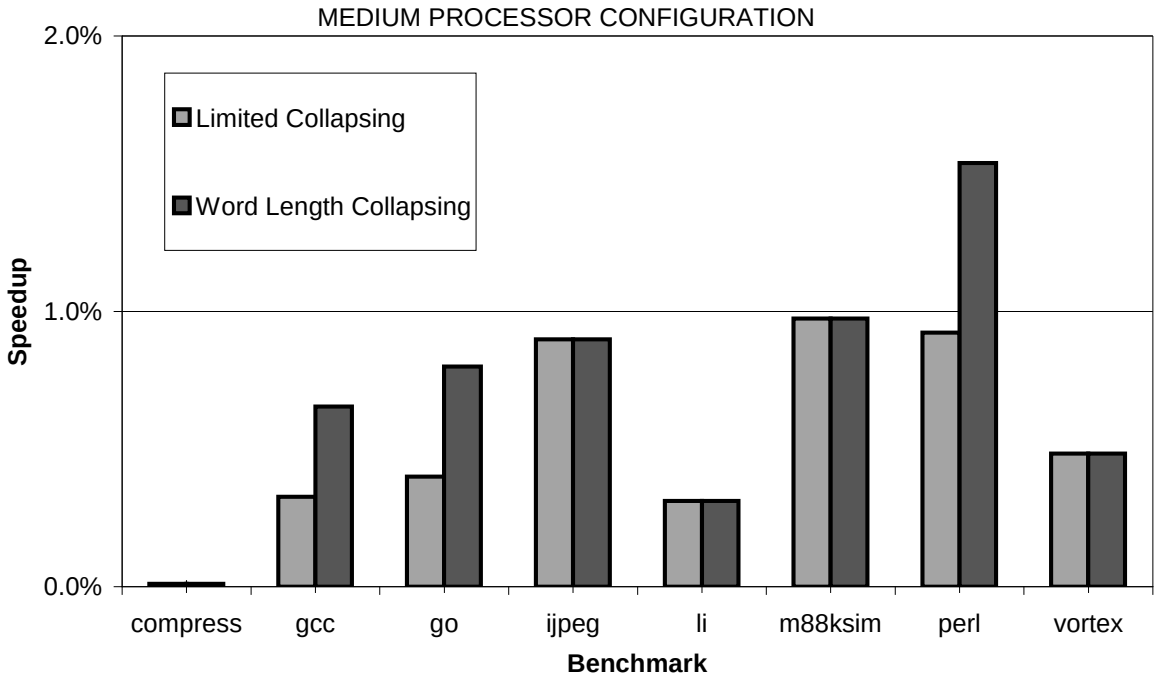


Figure 4-14 Speedup from constant propagation for the medium processor configuration.

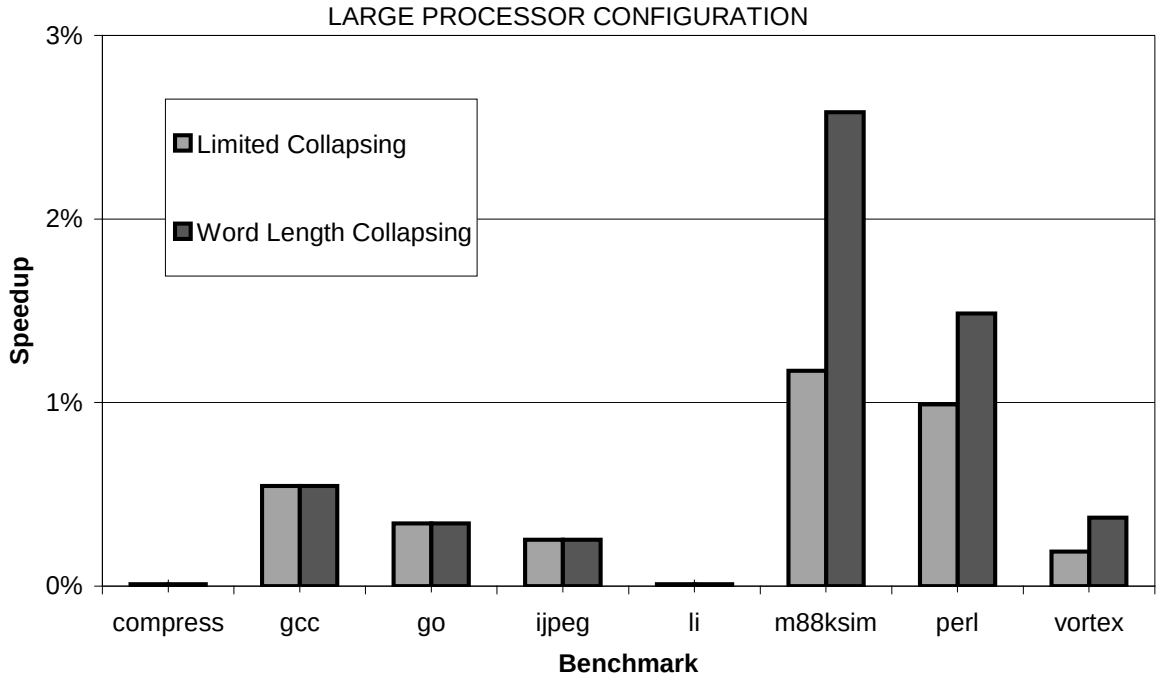


Figure 4-15 Speedup from constant propagation for the large processor configuration.

4.5.3 Instruction collapsing

Instruction collapsing removes data dependence delays by replacing instructions with new compound instructions. I consider three policies for collapsing instructions. The first case, labeled "Aggressive," is used to provide an upper bound for the potential of collapsing. It allows any dependent chain of fixed-point logical and simple arithmetic (addition and subtraction) operations to be collapsed as long as the total number of register operands is four or fewer. *Set* instructions and memory operations can be replaced by compound instructions. For memory operations the address computation is collapsed, but the actual memory operation is performed normally. Left shifts of immediate amounts of 3 or less are allowed anywhere in the computation chain. Instructions no longer needed after collapsing are not executed unless there is a branch misprediction (because we implement partial squashing of

traces) or an exception within the trace. This collapsing policy is likely not practical to implement as it would require extremely large instruction encoding space, complex and likely slow ALUs, and drastically increased register bandwidth.

The second case, labeled "Moderate," limits collapsed instructions to addition/subtraction of up to two register operands and one immediate operand. Set instructions and the address computation of memory operations can be collapsed. Either of the register operands can be shifted left by an immediate amount of 3 or less. This restriction limits the needed hardware support to a relatively simple arithmetic unit and requires no additional register ports or bypass logic. This new instruction can encode a number of possible chains of additions and left shifts. As before, instructions no longer needed after collapsing is performed are not executed unless there is a branch misprediction or an exception within the trace.

The third case, labeled "Conservative," is similar to the moderate case except that instructions no longer needed after collapsing are still executed. This simplifies the logic for branch misprediction recovery and precise exceptions. This also simplifies the logic for performing pre-processing because it does not require the determination of instructions that can be safely removed after collapsing.

There is significant opportunity to employ collapsing within traces. Figure 4-16 and Figure 4-17 show the number of instructions replaced by a new compound instruction for length 16 and length 32 traces respectively. The results for the length 16 traces are based on the medium processor configuration, and are not significantly different than those for the

small processor configuration. For length 16 traces, from approximately 5% to over 20% of instructions are replaced by compound instructions. With length 32 traces, there is slightly more opportunity for collapsing as there are larger groups of instructions to transform. For all the benchmarks except *ijpeg* and *m88ksim*, the moderate and conservative collapsing policies perform nearly as well as the aggressive policy. This suggests that the single new arithmetic unit and corresponding instruction are capable of capturing most of the cases of collapsing. In the case of *ijpeg* and *m88ksim*, most of the missed collapsing opportunities for the simpler approaches are chains of arithmetic operations that have more than two register operands.

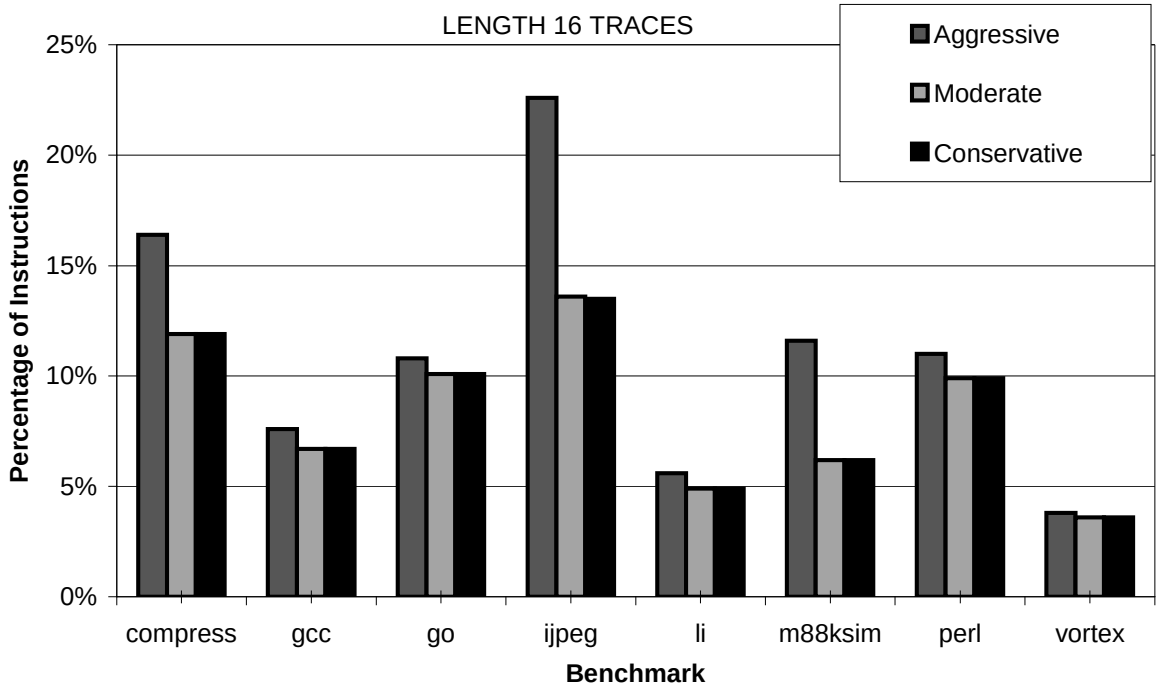


Figure 4-16 Percentage of instructions replaced by a compound instruction for length 16 traces.

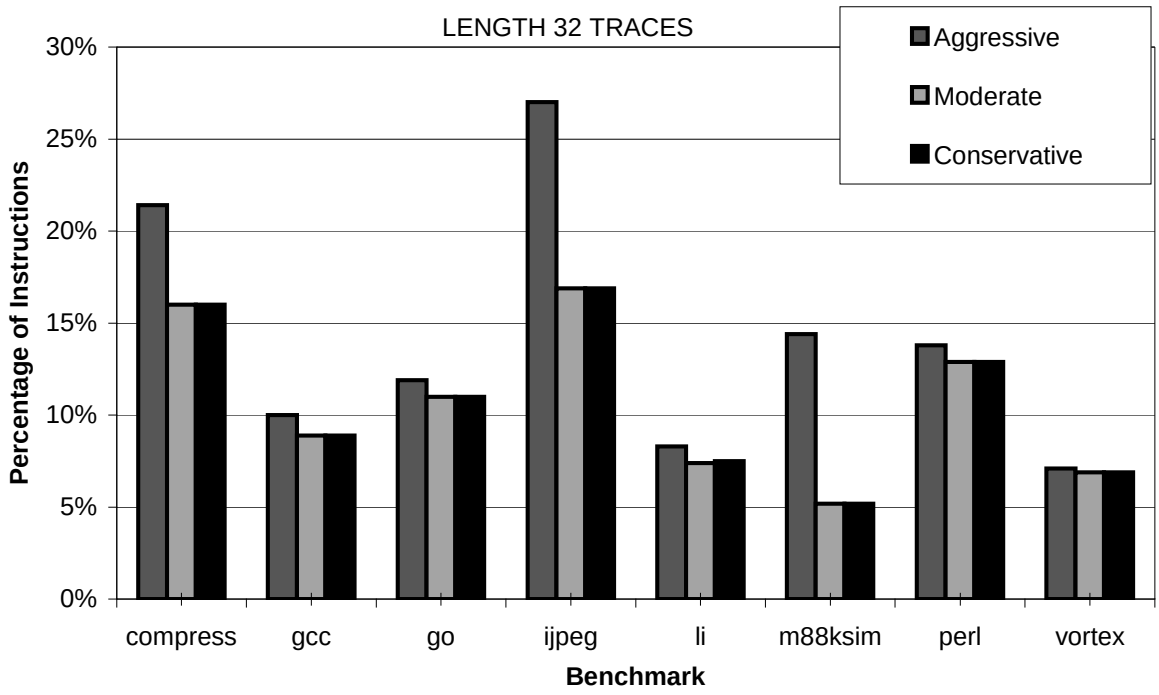


Figure 4-17 Percentage of instructions replaced by a compound instruction for length 32 traces.

For the aggressive and moderate policies, collapsing can cause instructions to be optimized away so that they no longer need to be executed. Figure 4-18 and Figure 4-19 show the percentage of instructions that are optimized away for length 16 and length 32 traces respectively. On average each compound instruction causes about two instructions to be removed. This leads to a significant percentage of the instructions being optimized away.

One of the ways that collapsing helps performance is by reducing the length of dependence chains, thereby reducing the latency of the dependence chain. Not all occurrences of collapsing achieve this. If a value needed at the end of the dependence chain is not available until after all other values are available, the collapsing will not reduce the latency. Looking at the example in Figure 4-2, if R2 is not available until much later than

R1, the compound instruction does not reduce the latency. In this example, R1 + 0x23 could be computed in advance and the value would be available when R2 comes available.

Figure 4-20 and Figure 4-21 show the percentage of instructions that issue early due to collapsing. These results only consider the cases where the latencies of dependence chains are reduced. They do not consider the benefits of optimizing away instructions or whether the result of a dependent chain is on the critical path. An instruction is counted as issuing early if it is a compound instruction and the last operand to come available was not one of the original input operands to the replaced instruction (the instruction at the end of the dependence chain). The majority (64% on average of length 16 traces and 75% on average for length 32 traces) of instructions replaced by compound instructions do issue early due to collapsing.

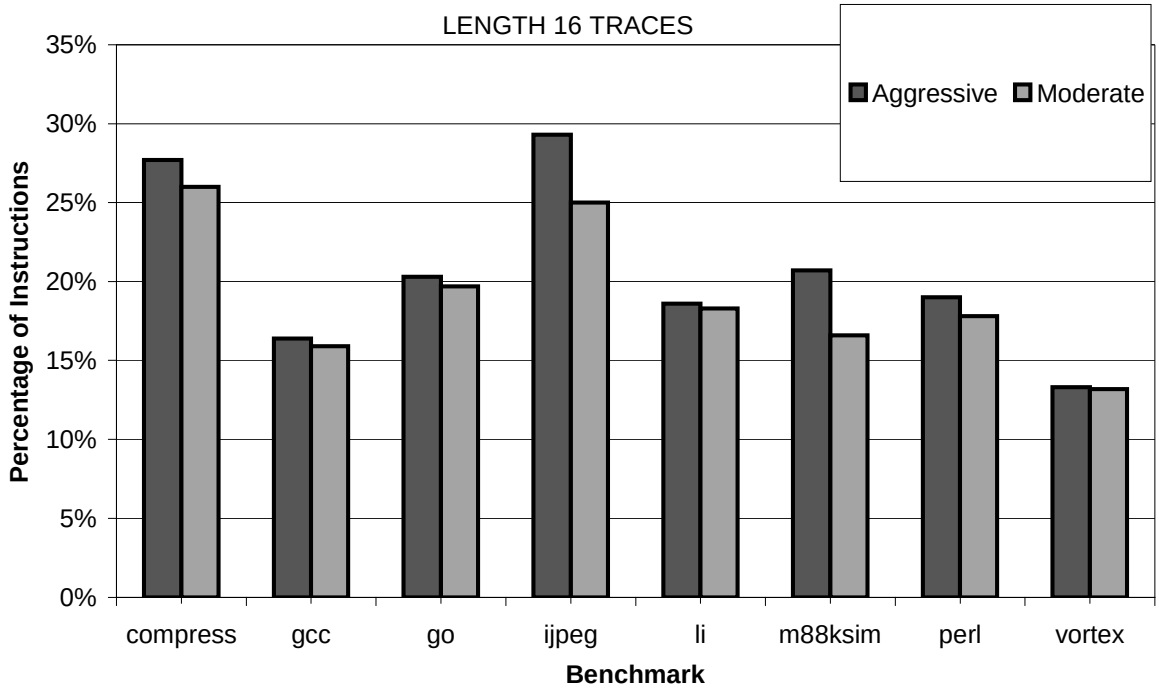


Figure 4-18 Percentage of instructions optimized away by collapsing for length 16 traces.

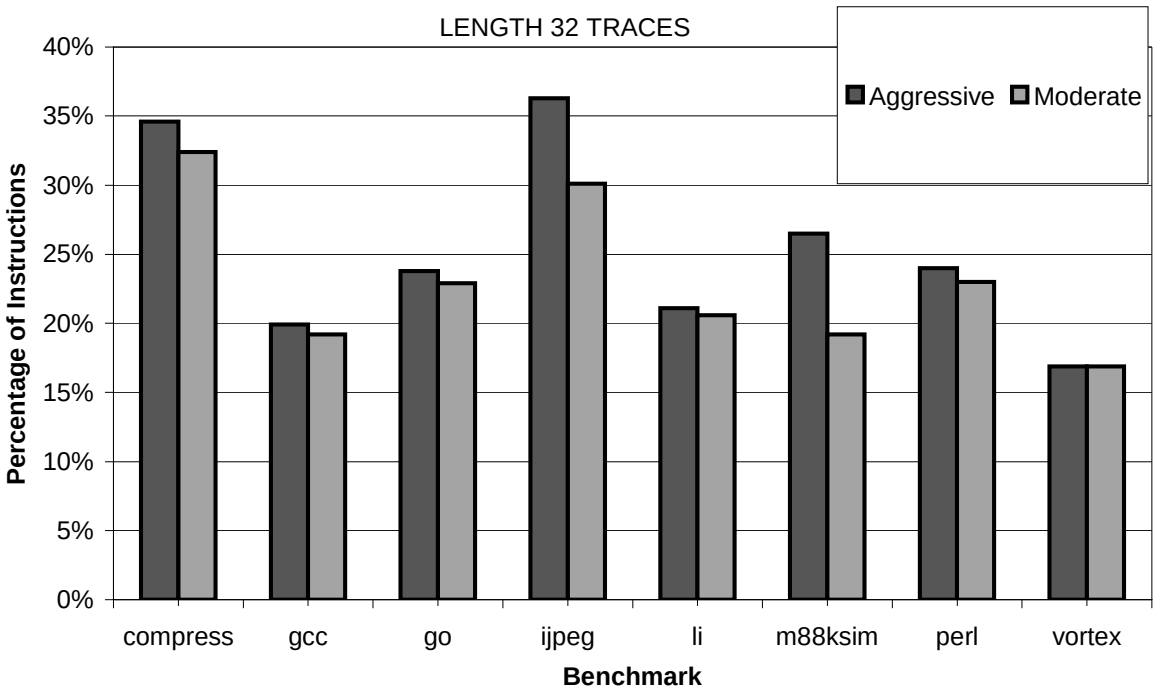


Figure 4-19 Percentage of instructions optimized away by collapsing for length 32 traces.

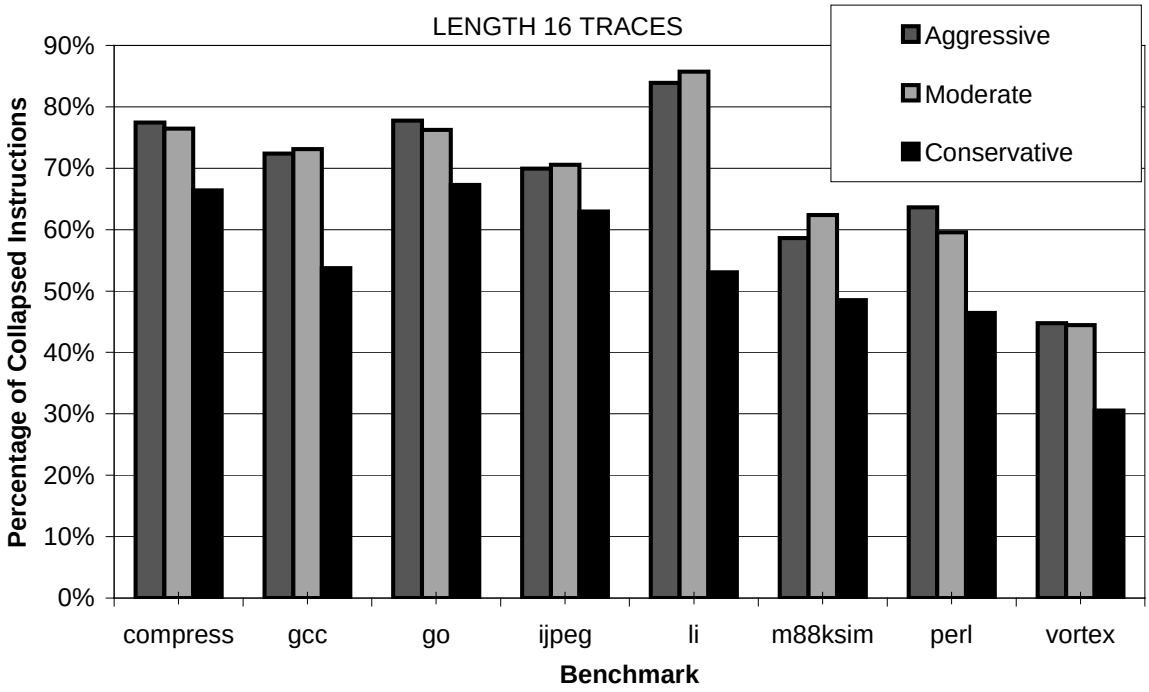


Figure 4-20 Percentage of collapsed instructions that issue early due to collapsing for length 16 traces.

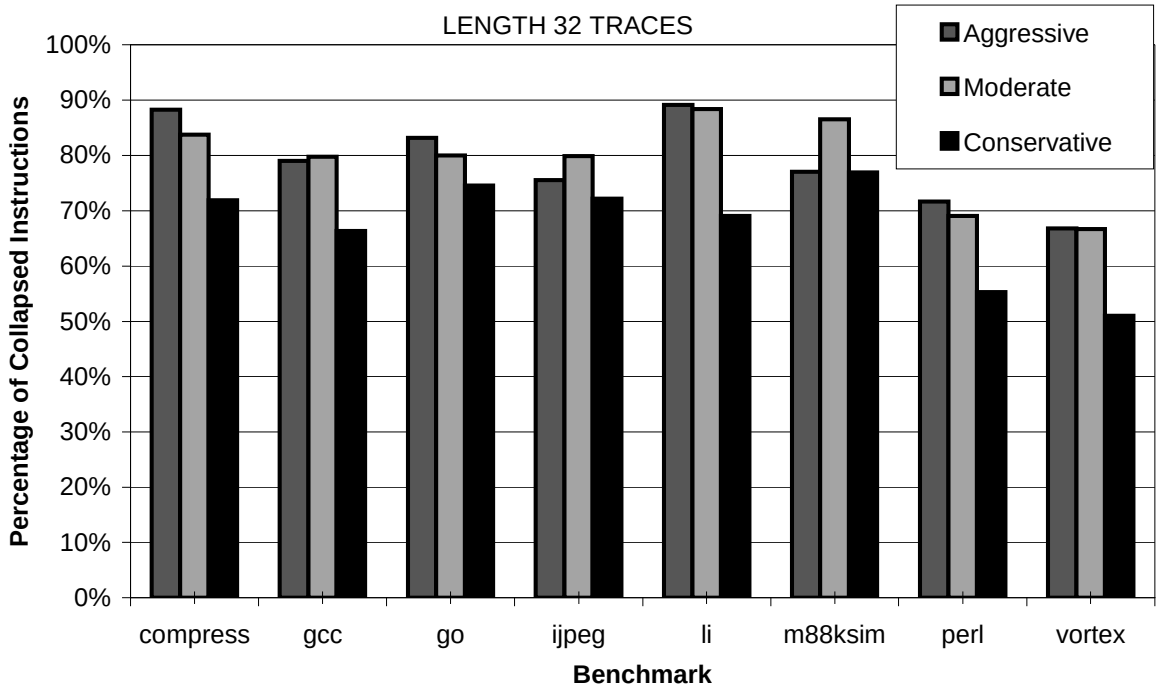


Figure 4-21 Percentage of collapsed instructions that issue early due to collapsing for length 32 traces.

Overall speedup is the true measure of a performance optimization. The collapsing optimization effects a significant number of instructions; more importantly it leads to a significant overall speedup. Figure 4-22 through Figure 4-29 show the speedup from collapsing for the eight SPECint95 benchmarks. The aggressive collapsing policy leads to a 4% to 14% speedup for most of the benchmarks, a significantly higher speedup is seen for the benchmark *m88ksim*. The average speedup from the aggressive collapsing is 10% (with a harmonic mean of 9%).

The moderate collapsing comes very close to the performance of the aggressive collapsing, achieving an average speedup of 9% (with a harmonic mean of 8%). Many of the benchmarks have comparable degrees of collapsing for aggressive and moderate collapsing, so comparable performance is expected. But, even for the benchmark *ijpeg* and *m88ksim*

(where the aggressive collapsing was able to transform significantly more instructions), the performance difference is much smaller than the difference in the number of instructions transformed.

Conservative collapsing has on average only a 4% speedup. Removing the unnecessary instructions to free up issue bandwidth has a significant impact. The impact is slightly less for the large processor configuration, where there is slightly more flexibility in terms of issue bandwidth. Scheduling can address the problems of conservative collapsing, as will be seen in the next section.

Collapsing provides a nearly fixed benefit on top of the existing processor configuration. In general the smaller processor configurations see a larger percentage speedup from collapsing. The difference is roughly proportional to the difference in absolute performance.

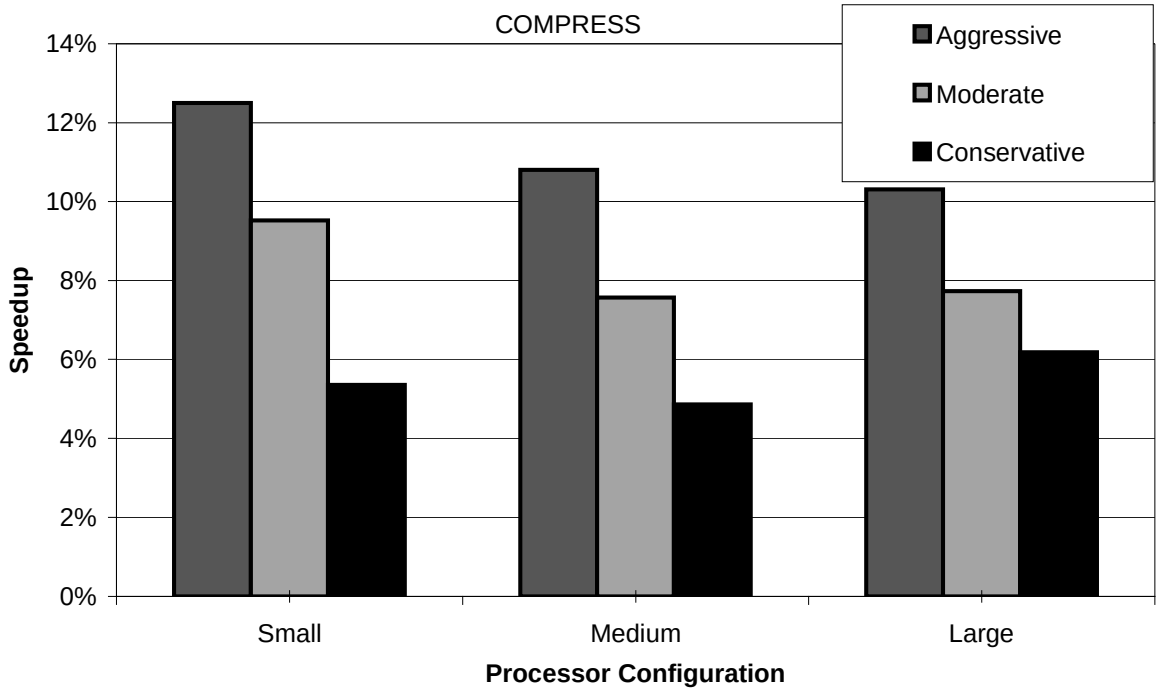


Figure 4-22 Overall speedup from the collapsing optimization for *compress*.

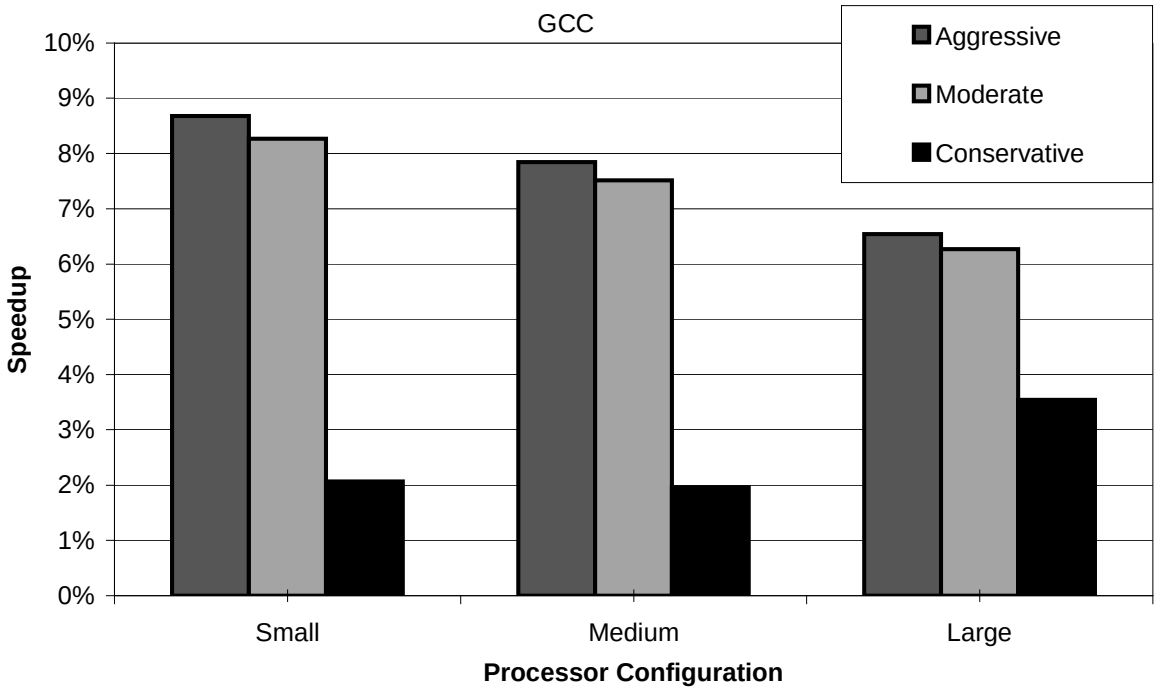


Figure 4-23 Overall speedup from the collapsing optimization for *gcc*.

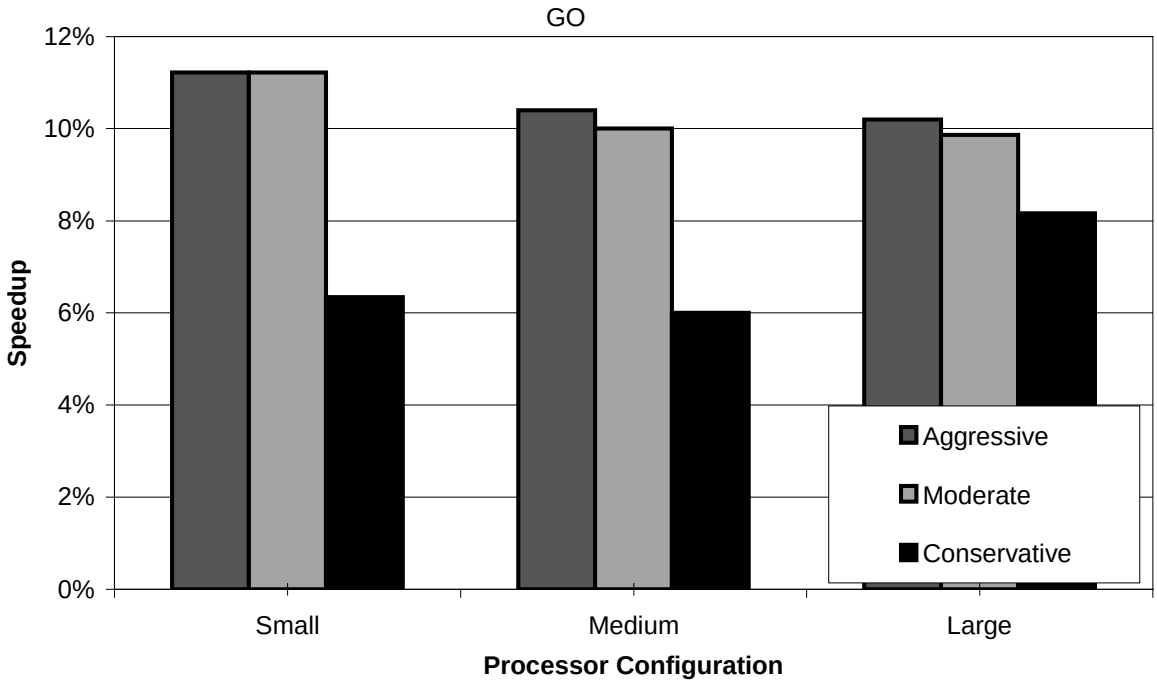


Figure 4-24 Overall speedup from the collapsing optimization for *go*.

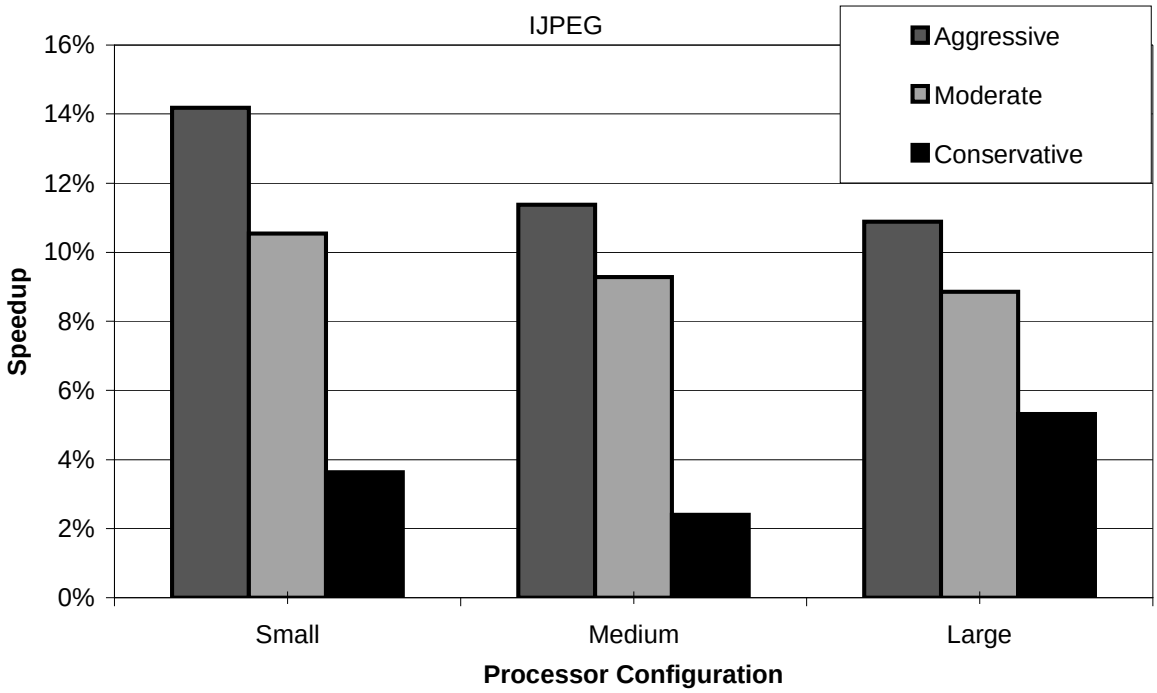


Figure 4-25 Overall speedup from the collapsing optimization for *ijpeg*.

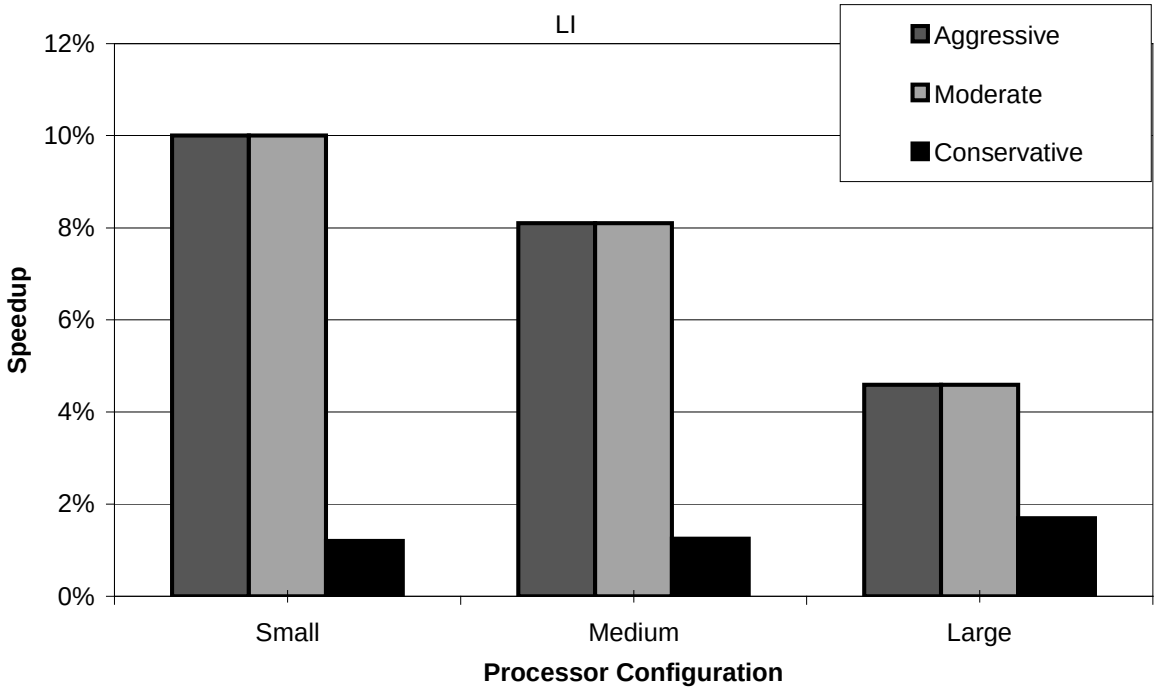


Figure 4-26 Overall speedup from the collapsing optimization for the benchmark *li*.

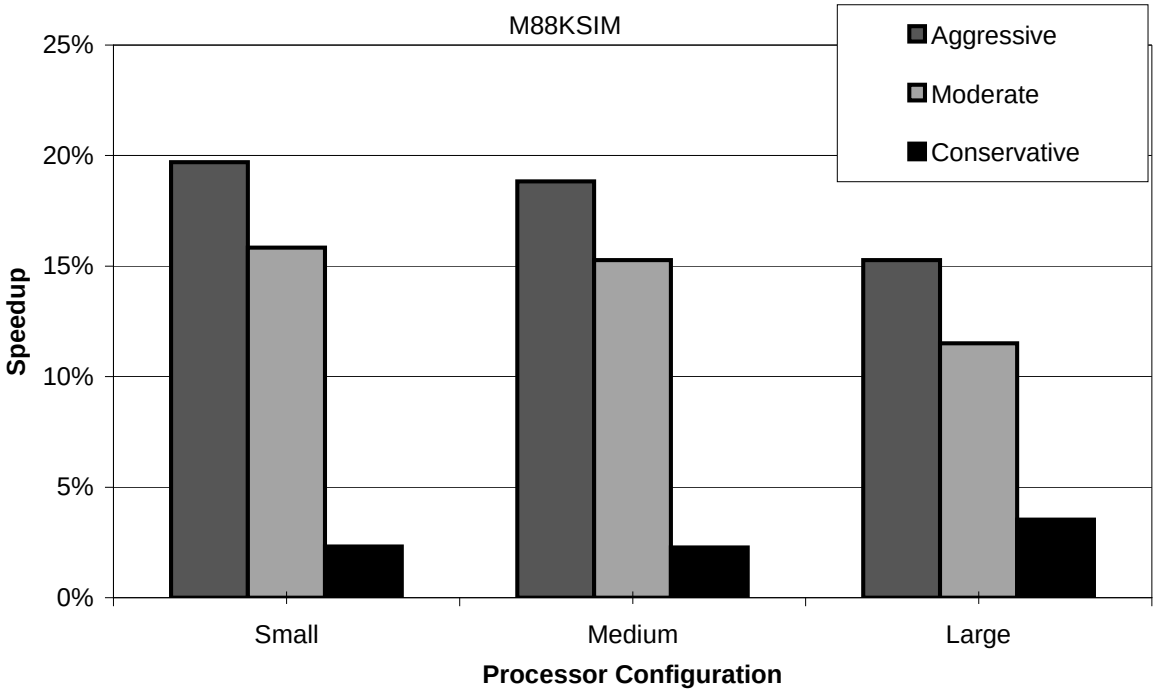


Figure 4-27 Overall speedup for the collapsing optimization for *m88ksim*.

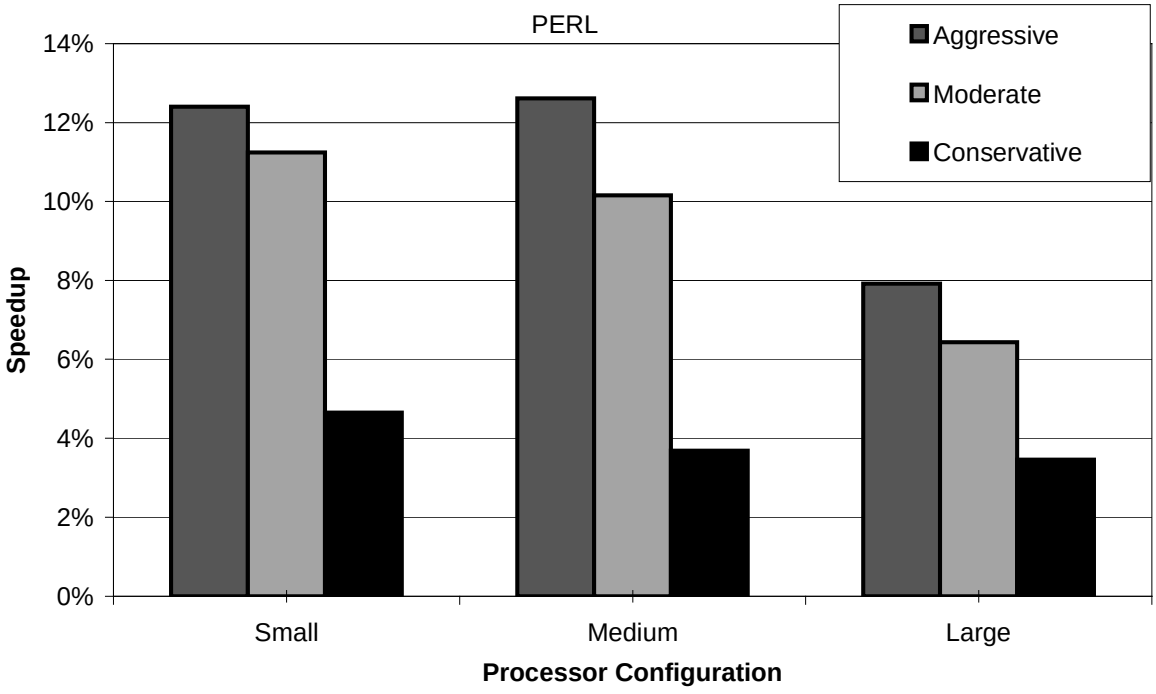


Figure 4-28 Overall speedup from the collapsing optimization for *perl*.

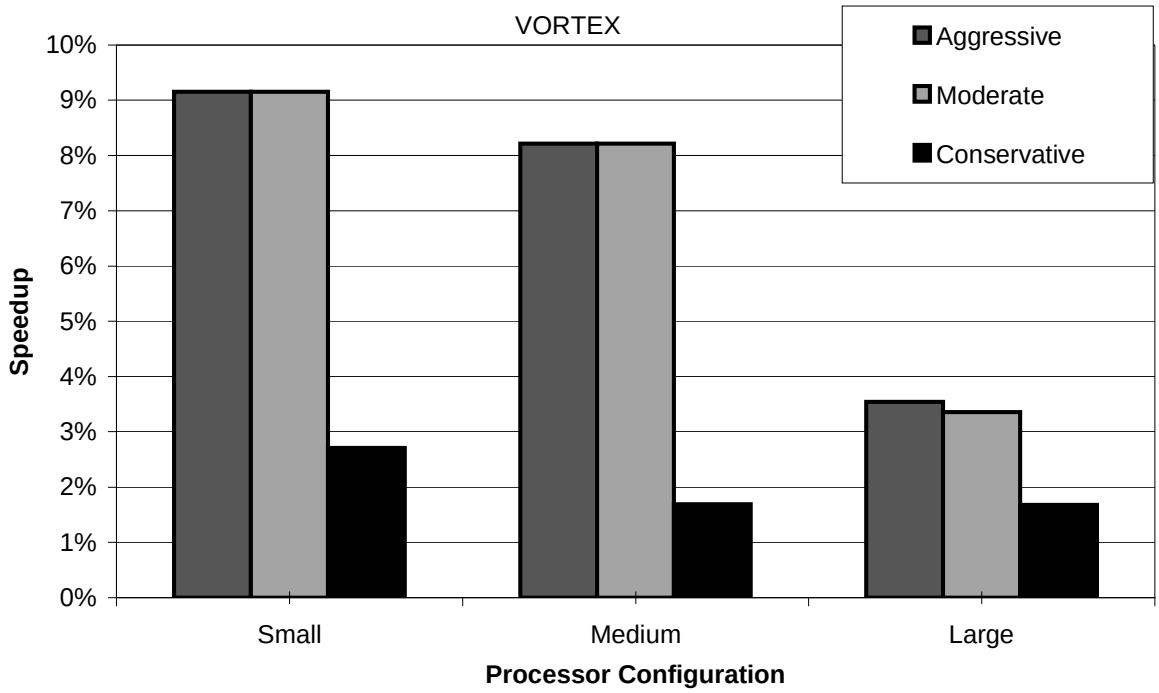


Figure 4-29 Overall speedup from the collapsing optimization for *vortex*.

4.5.4 Combining pre-processing optimizations

In general, interesting things happen when optimizations are combined. The interactions can be complicated, with one optimization sometimes diminishing the benefit of another, or enhancing the benefit of another. Often, the resulting performance is considerably less than the sum of the parts, frequently with the result being comparable to the largest part. Sometimes, the result is equal to or greater than the sum of the parts. Both cases are seen when pre-processing optimizations are combined.

Figure 4-30 through Figure 4-37 show the speedup from combining the pre-processing optimizations for the SPECint95 benchmarks. For each benchmark, results are given for all three processor configurations. For each configuration, two sets of results are given; one

where the moderate collapsing policy is used and one where the conservative collapsing policy is used. Both collapsing policies add a single new compound instruction that uses only two register source operands. The moderate policy removes unnecessary instructions after collapsing while the conservative policy executes all instructions. The results show the breakdown for first adding collapsing, then adding the scheduling optimization (the combined heuristic) and finally adding the limited constant propagation optimization (makes use of existing constant fields in instructions as opposed to adding word length immediate operands).

When the moderate collapsing policy is used, there is minimal benefit from incorporating the other optimizations. Moderate collapsing removes a significant number of instructions, reducing the contention for issue bandwidth and thereby reducing the potential benefit of scheduling. Moderate collapsing also captures much of the behavior of the constant propagation optimization. A chain of computation with immediate operands can be collapsed into the instruction that uses the result, removing the dependences and unnecessary instructions in the same way as constant propagation.

The scheduling and constant propagation optimizations show more benefit when applied with conservative collapsing than when applied alone. Conservative collapsing increases the amount of parallelism by breaking dependence chains, this increases the contention for issue bandwidth. Scheduling can effectively hide much of the extra contention for issue bandwidth seen with conservative collapsing as compared to moderate collapsing. Together, conservative collapsing and scheduling often perform nearly as well as the case where

moderate collapsing is used. The computation removed by constant propagation helps further to reduce the increased contention for issue bandwidth from conservative collapsing.

Together, the pre-processing optimizations lead to a 5% to 20% speedup in performance, with an average speedup of 11% for moderate collapsing and 9% for conservative collapsing (with harmonic means of 9% and 8%). The collapsing feature of removing unnecessary computation offers some benefit, but it is small and depending on the implementation complexity it may not be worth implementing. There are also significant opportunities to incorporate additional optimizations into the pre-processing mechanism to produce even greater performance benefits.

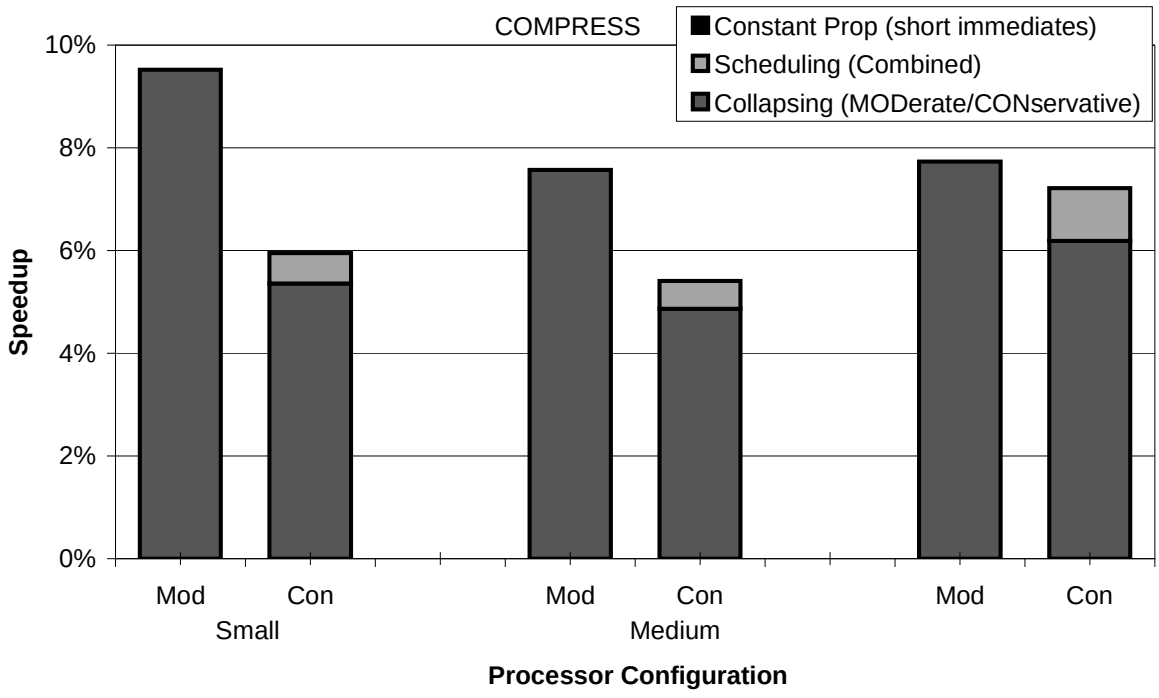


Figure 4-30 Speedup from combined pre-processing optimizations for *compress*.

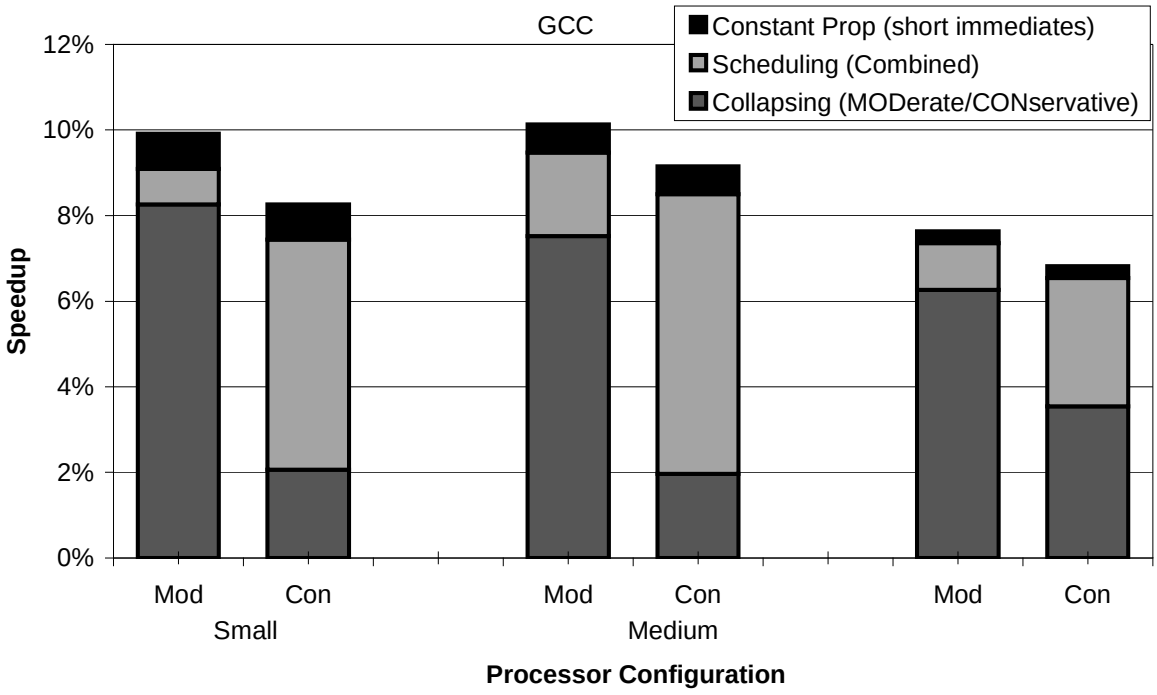


Figure 4-31 Speedup from combined pre-processing optimizations for *gcc*.

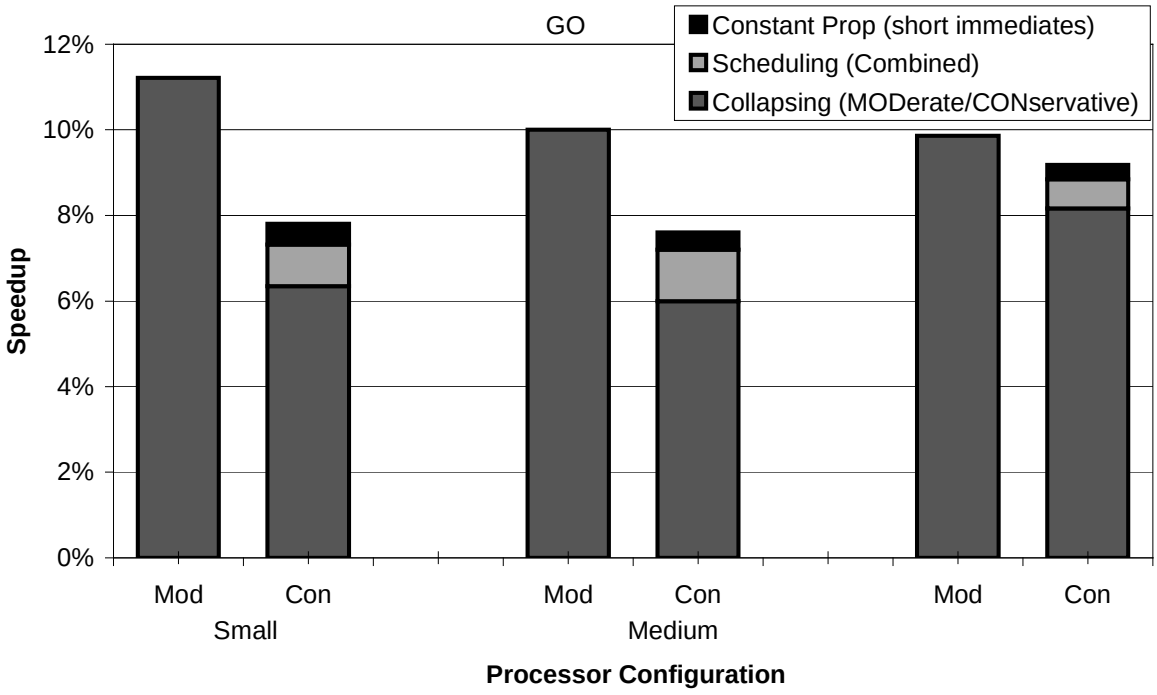


Figure 4-32 Speedup from combined pre-processing optimizations for *go*.

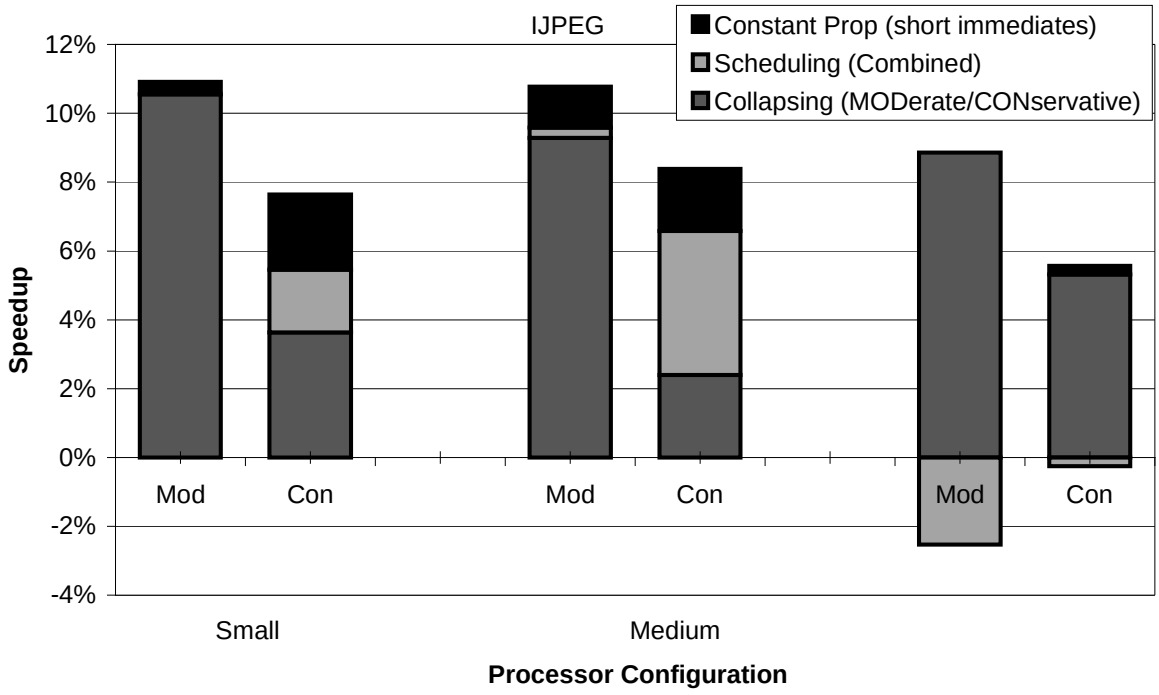


Figure 4-33 Speedup from combined pre-processing optimizations for *ijpeg*.

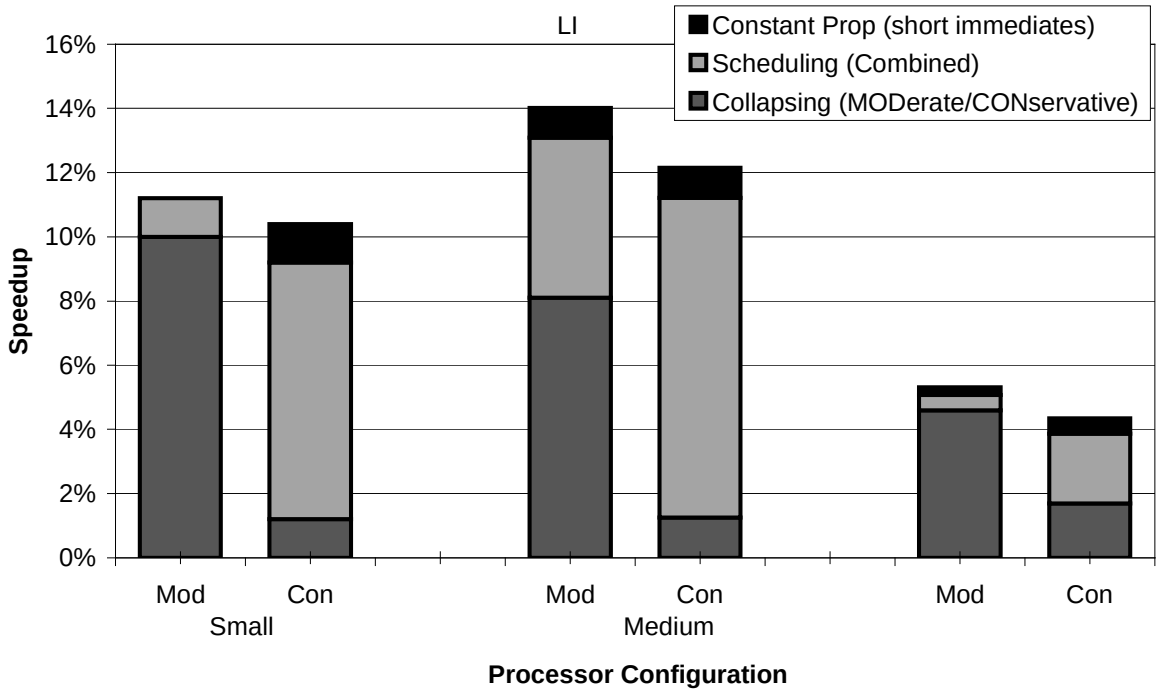


Figure 4-34 Speedup from combined pre-processing optimizations for *li*.

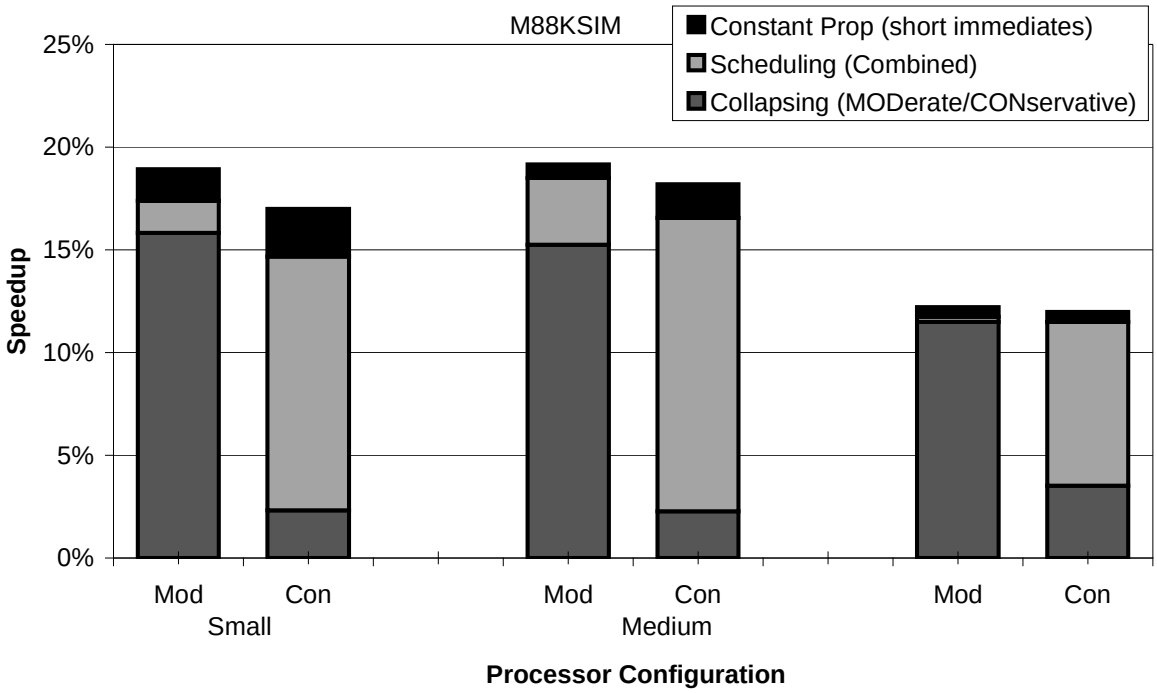


Figure 4-35 Speedup from combined pre-processing optimizations for *m88ksim*.

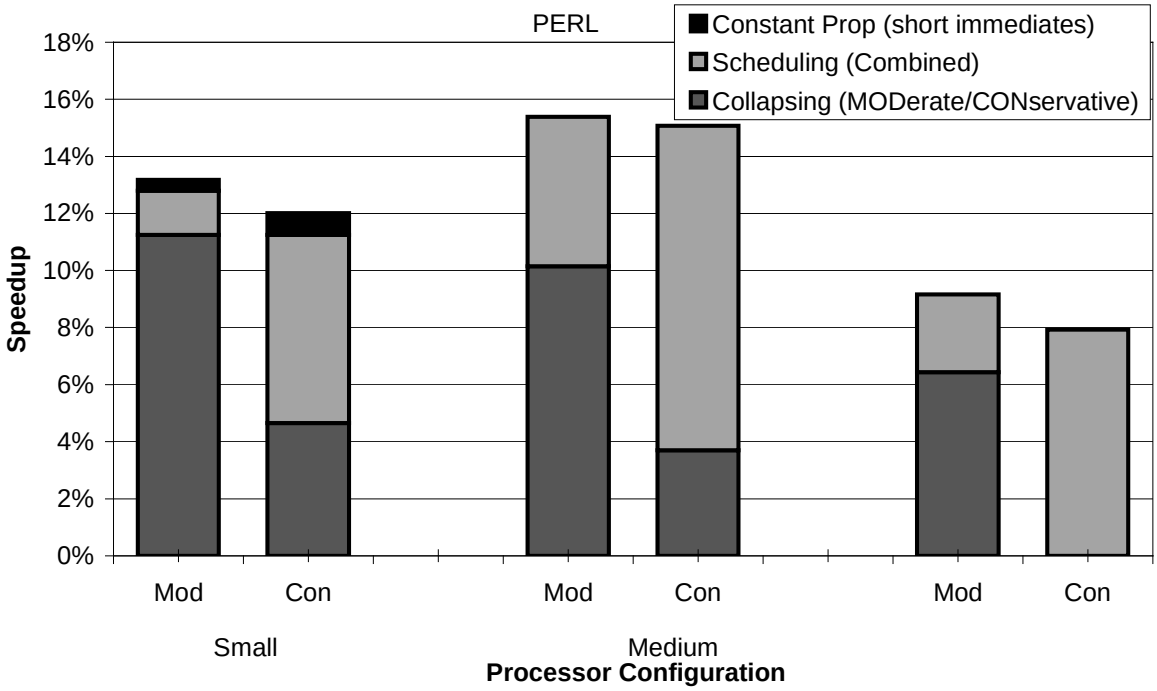


Figure 4-36 Speedup from combined pre-processing optimizations for *perl*.

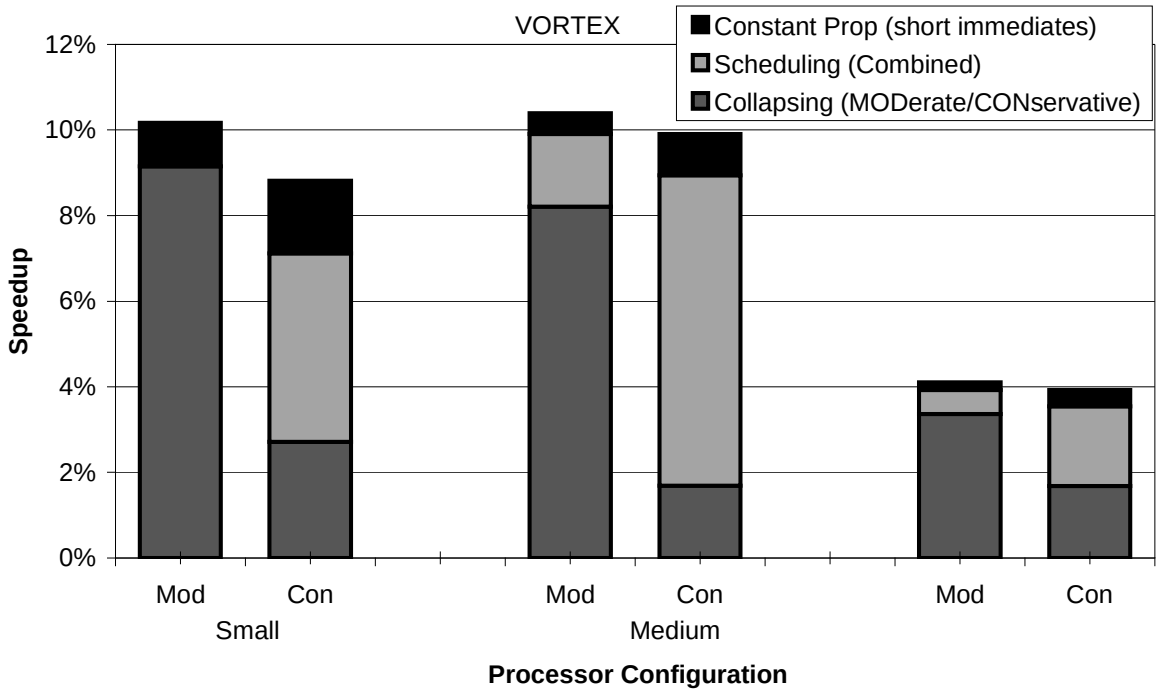


Figure 4-37 Speedup from combined pre-processing optimizations for *vortex*.

4.6 Combining Pre-Construction and Pre-Processing

Pre-construction and pre-processing both take advantage of larger units of work, traces, to increase the performance of different parts of the processor. Pre-construction attempts to increase the instruction supply bandwidth while pre-processing attempts to increase the instruction execution bandwidth. I have shown that pre-construction and pre-processing both offer performance improvements individually. In this section I explore what happens when they are applied together.

There are two ways that pre-construction and pre-processing can be integrated. First, the mechanisms can both be added to the processor with no special interaction. As long as the trace cache can hold unoptimized traces, the two mechanisms can be incorporated without

any interference. Second, the mechanisms can be integrated in such a way that the traces produced by the pre-construction engine are immediately fed to the pre-processing engine to be optimized.

4.6.1 Conservative integration

Incorporating both pre-construction and pre-processing without any interaction between the mechanisms will be referred to as *conservative* integration. Implementing this design is straightforward. Both mechanisms are incorporated exactly as they were described in early sections. The pre-construction engine produces traces that are placed into a pre-construction buffer. Whenever an unoptimized trace is executed, the task of pre-processing it is started during the execution and completed afterwards by the pre-processing engine, which places the optimized trace into the trace cache. When an optimized trace is placed into the trace cache, it overwrites an unoptimized version in the cache if present, and invalidates any corresponding entry in the pre-construction buffer if present.

4.6.2 Aggressive integration

Incorporating both pre-construction and pre-processing with the traces generated from pre-construction being pre-processed will be referred to as *aggressive* integration. There are a number of issues involved in implementing this design. Much of the work of pre-processing is integrated into the execution engine and performed during the execution of the trace. If traces from the pre-construction engine are to be pre-processed, this approach will not work; separate execution resources are necessary to perform the computational component of pre-processing. Then there are the issues of the bandwidth and latency of these

resources as well as the bandwidth and latency of the logic to actually transform the instructions within a trace.

In order to determine the potential of the aggressive integration approach, an ideal implementation is considered. For this implementation, all traces placed in the pre-construction buffer are pre-processed with zero latency. This establishes an upper bound on how much performance can be achieved by pre-processing the traces created by the pre-construction engine. The actual performance achieved would be somewhere between the performance of the conservative integration and this ideal aggressive integration.

The results (given in the next section) show that there is negligible benefit in going from the conservative approach to the aggressive approach. Therefore, it does not make sense to apply any extra resources to perform the pre-processing of traces generated by the pre-construction engine.

4.6.3 Performance of integrated frontend

In this section I consider the overall speedup from applying pre-construction and pre-processing together. I only consider the four benchmarks that saw the most significant benefit from pre-construction: *gcc*, *go*, *perl* and *vortex*. Either a 512 entry trace cache is used alone, or a 256 entry trace cache and a pre-construction engine with a 256 entry pre-construction buffer are used. For pre-processing, the conservative collapsing policy is used in conjunction with instruction scheduling and constant propagation. Constant propagation is limited to immediate fields that fit within the normal instruction encoding.

Figure 4-38 through Figure 4-41 show the speedup from pre-construction and pre-processing. Results are shown for pre-construction and pre-processing in isolation, then results are presented for pre-construction and pre-processing integrated with both the conservative and aggressive approaches. A reference value, showing the simple sum of the speedups from pre-construction and pre-processing optimizations in isolation, is given for comparison.

The individual speedups from the optimizations are in line with the results seen earlier. With a large trace cache and the relatively small benchmarks, the speedup from pre-construction is relatively small (1% to 5% with an average of 2%). Pre-processing leads to more substantial speedups, from 4% to 15% speedup, with an average of 9%.

The speedup from applying pre-construction and pre-processing together is consistently above the sum of the individual speedups. The average speedup achieved is 12% (with a harmonic mean of 9%). Considering only the small and medium processor configurations, the average speedup goes up to 14% (with a harmonic mean of 13%). This speedup is not surprising, as one optimization helps the frontend of the processor and the other helps the backend of the processor. Either optimization alone increases the potential bandwidth of one of the pipelines, but the benefit is limited by the other pipeline becoming more of a bottleneck. Increasing the potential bandwidth of both the frontend and backend pipelines together leads to a more substantial increase in overall bandwidth of the processor.

The additional benefit from applying the optimizations together makes the pre-construction optimization more appealing. Pre-processing offers sufficient speedup alone to

warrant the complexity of implementing it. Pre-construction offers less speedup alone, although it does help robustness to deal with events like context switches. When combined with pre-processing the benefit of pre-construction is more substantial. This coupled with the added robustness makes pre-construction appealing.

The benefit of the aggressive combination of pre-construction and pre-processing over the conservative combination is insignificant (as shown in Figure 4-38 through Figure 4-41). The extra complexity to pre-process the traces produced by the pre-construction engine is not warranted.

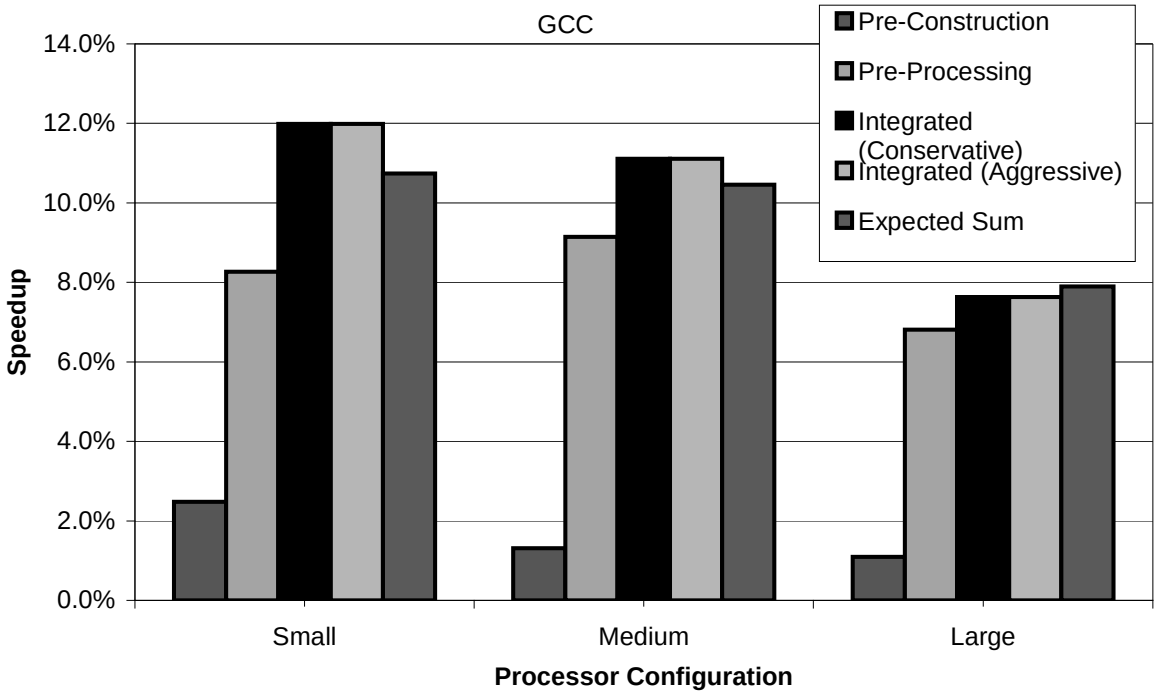


Figure 4-38 Speedup from pre-construction and pre-processing for *gcc*.

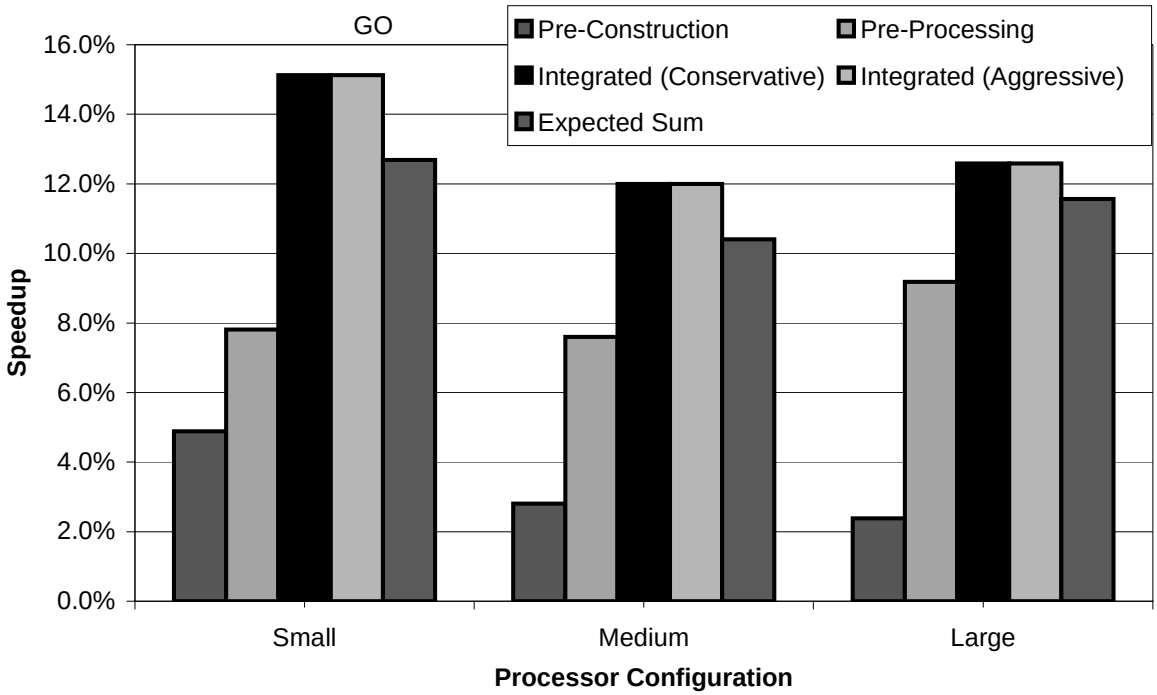


Figure 4-39 Speedup from pre-construction and pre-processing for *go*.

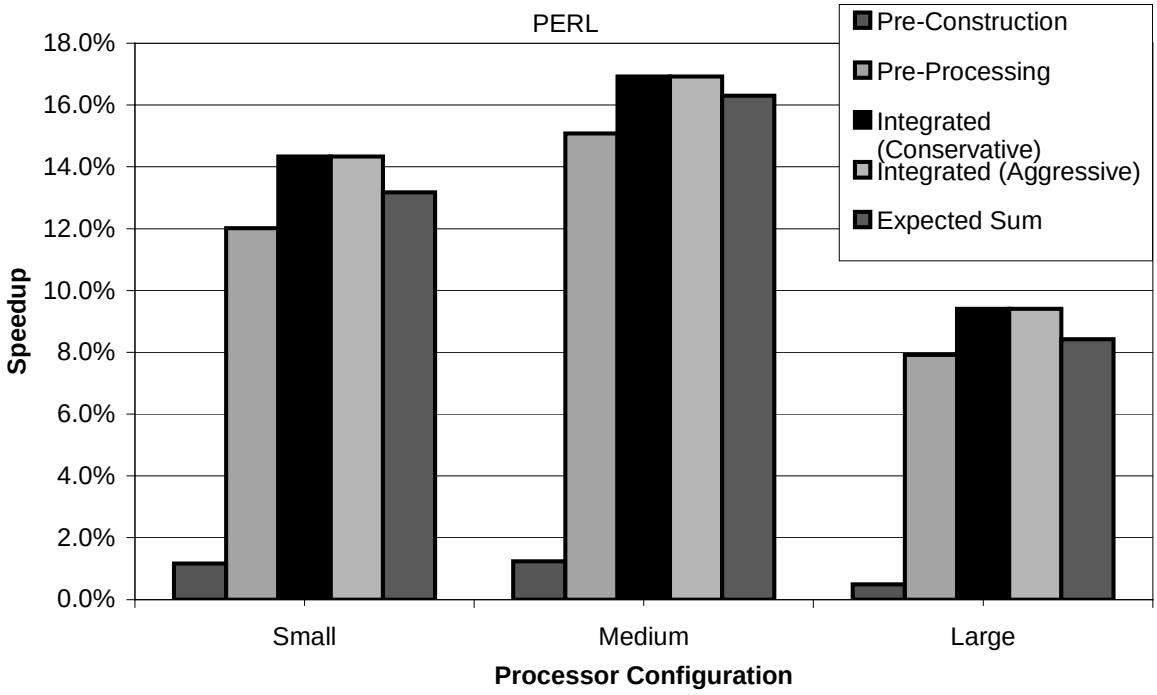


Figure 4-40 Speedup from pre-construction and pre-processing for *perl*.

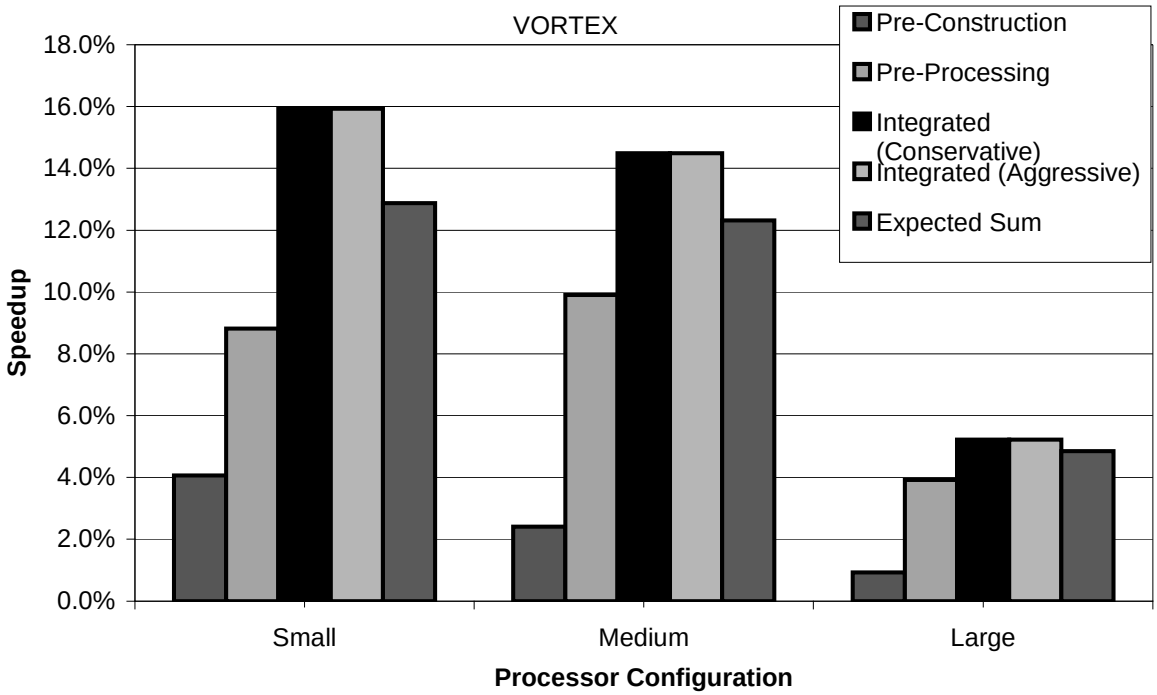


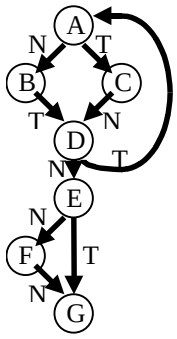
Figure 4-41 Speedup from pre-construction and pre-processing for *vortex*.

Chapter 5
Next-Trace Prediction

In pipelined processors, instructions are fetched, decoded and executed over a series of pipeline stages. In order to keep the pipeline full it is necessary to predict the behavior of branch instructions before they are even decoded. This involves learning the location and target of branch instructions and predicting the behavior of conditional branch instructions. Using prediction, the instruction fetch unit can continue feeding instructions into the pipeline even though branch instructions are identified and executed later in the pipeline. Later stages of the pipeline, where the decoding and execution of branches is performed, provide information to the fetch unit to recover from incorrect predictions and provide information to aid in making future predictions. In trace processors the task of predicting the behavior of branches is even more complicated than normal processors because the fetch mechanism attempts to sustain much higher bandwidth.

Associated with the trace cache is a trace fetch unit, which fetches a trace from the cache each cycle. To do this in a timely fashion, it is necessary to predict what the next trace will be. A straightforward method, and the one used for initial work with trace caches [40][48], is to predict simultaneously the multiple branches within a trace. Then, armed with the last PC of the preceding trace and the multiple predictions, the fetch unit can access the next trace. In the example CFG shown in Figure 5-1, if trace 1 -- ABD -- is the most recently fetched trace,

and a multiple branch predictor predicts that the next three branch outcomes will be T,T,N, then the next trace will implicitly be ACD.



Possible Traces:

1. ABD
2. ACD
3. EFG
4. EG

Figure 5-1 Example CFG

I propose a different approach to next-trace prediction — traces are treated as the basic unit and the processor explicitly predicts sequences of traces. For example, referring to the above list of traces, if the most recent trace is trace 1, then a next-trace predictor might explicitly output “trace 2.” The individual branch predictions T,T,N, are implicit.

I propose and study *next-trace predictors* that collect histories of trace sequences and make predictions based on these histories. This is similar to conditional branch prediction where predictions are made using histories of branch outcomes. However, each trace typically has more than two successors, and often has many more. Consequently, the next-trace predictor keeps track of sequences of trace identifiers, each identifier containing multiple bits. I propose a basic predictor and then add enhancements to reduce performance losses due to cold starts, procedure call/return pairs, and interference due to aliasing in the prediction table.

5.1 Incorporating Next-Trace Prediction

5.1.1 The predictor drives the pipeline

The next-trace predictor observes the sequence of traces and attempts to learn the behavior so that it can make accurate predictions. It provides a stream of predictions that drive the fetching of traces from the trace cache, which in turn drives the rest of the processor (see Figure 5-2). The predictor contains a tight feedback loop: based on the last prediction, it makes the next prediction. The predictor receives information from the other frontend mechanisms to throttle it when the frontend pipeline is stalled or when the instruction window is full. The predictor also receives information from the backend pipeline regarding whether or not a prediction was correct.

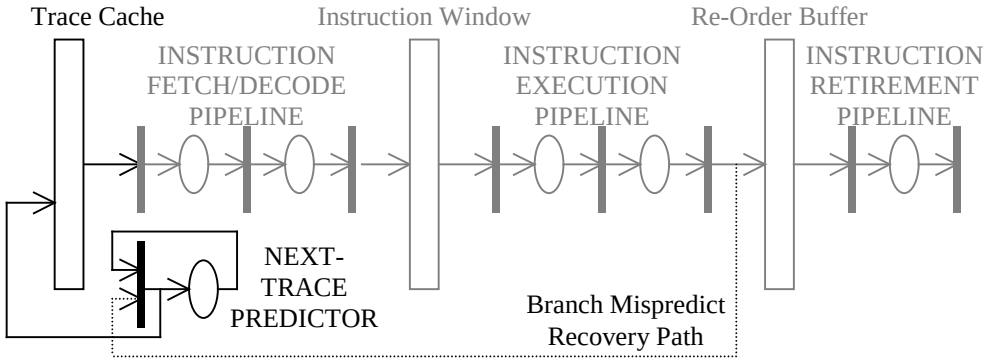


Figure 5-2 Incorporating next-trace prediction.

Next-trace prediction (working with the trace cache) implicitly takes care of the functions of identifying where control-transfer instructions are, what the targets of the instructions are, and the direction of conditional branches. In traditional processors there are often two mechanisms, a predictor of conditional branches and a branch target buffer (BTB) [54] for identifying branches and remembering their targets.

5.1.2 Trace granularity vs. instruction granularity

Using trace-level granularity for control prediction affects the processor in a number of ways. Three of the most significant impacts are discussed in this section. First, it matches the control prediction bandwidth with the fetch bandwidth. Second, it makes transitioning to and from instruction granularity and trace granularity more difficult. Finally, the control prediction accuracy becomes dependent on trace selection heuristics.

Having the predictor work in the same units as the fetch mechanism matches the bandwidth of the two steps. If a multiple branch predictor is used, traces must be restricted to containing a maximum number of branch instructions in order to guarantee that the predictor can keep up with the fetching of traces. This restriction is no longer needed if the predictor uses traces.

A liability of the predictor working at trace granularity is that it increases the complexity of transitioning between trace granularity and instruction granularity in the frontend. The frontend primarily works at trace granularity, but there are two circumstances when it needs to transition to instruction granularity. First, if the next-trace predictor can not generate a valid prediction for the next trace, a trace must be generated at an instruction granularity. Second, when a branch misprediction is detected in the middle of a trace, a valid alternate trace with the same beginning must be generated at an instruction granularity. The latter can be mostly, but not completely, avoided with alternate trace prediction (see Section 5.7). The next-trace predictor does not provide information about individual branches, so a separate branch predictor is needed to provide predictions for instruction granularity sequencing.

There is no clean interface between the trace granularity predictor and instruction granularity predictor to allow them to share information. Transitioning back to the trace granularity prediction can occur only at trace boundaries.

The performance of a predictor based on trace granularity sequencing is dependent on the trace selection heuristic. The average size of traces, determined by the selection heuristic, affects the unit size of the predictor, which in turn affects its performance. Less obvious, but at least as important, is the issue of trace alignment. As discussed in Section 2.2.1.1, there can be multiple trace sequences for the same dynamic sequence of code depending on the starting point of the first trace. The alignment is a function of the code executed prior to the region of code in question. The alignment therefore contains some information about past program behavior. This information can potentially be useful to prediction, if there is a correlation between the behavior that determined the alignment and the behavior of later branches. An example of useful alignment information is the case of a loop where the trace alignment can implicitly encode some information regarding the iteration count of the loop. This information can also be detrimental however, as the predictor must learn the same dynamic path based on different alignments.

5.2 The Basics of Control Prediction

5.2.1 Fundamentals of prediction

Control predictors work by observing how branches have behaved in the past to predict how they will behave in the future. The simplest predictor maintains a table that records a

branch's behavior the last time it was encountered and predicts that a branch will perform similarly in the future. There are two fundamental techniques to increase the accuracy of prediction: hysteresis and correlation.

5.2.1.1 Hysteresis

Hysteresis is applied to mask rare or anomalous behavior. If a branch has been consistently behaving a certain way and is then observed behaving differently once, it may be better to continue predicting the original behavior until it is clear that the branch is changing its behavior. Hysteresis is implemented by incorporating a saturating counter along with each of the entries remembering previous branch behavior. When the prediction is correct the counter is incremented. When the prediction is not correct, the saturating counter is decremented (it need not be incremented and decremented by the same amount). In the case that the prediction is not correct and the counter is at zero, the prediction is replaced. The degree of hysteresis is important. Too much hysteresis and the predictor is slow to adapt to new behavior; too little and the predictor gets confused by anomalous behavior. A 1-bit or 2-bit counter is often appropriate. For predicting the direction of branches, the functionality of having one bit to encode the direction of the branch and a 1-bit hysteresis counter are combined into a single 2-bit saturating counter that counts the number of times a branch is taken.

5.2.1.2 Correlation

Correlation increases prediction accuracy by making predictions based on more program information, most commonly other branch information. The predictor learns how a branch

behaves for a number of different contexts and makes predictions specific to a context. The two most common types of correlation are local history and global history. With local history, the outcomes of the past few executions of an individual branch (or a small group of branches) are remembered. Predictions are based on how a branch has previously behaved following the same specific sequence of outcomes. Local history tends to exploit repetitive, cyclical behavior. With global history, the behavior of control instructions that occurred previously in the dynamic instruction stream are remembered, and predictions are based on the history observed for the same dynamic sequence.

There are many forms of correlation possible, but I will focus on global history. Global history predictors have been shown to have very good performance and have become the standard approach for current high performance control predictors.

Correlation can significantly increase performance, as there are often recognizable patterns in the behavior of programs. In some cases though, correlation hurts performance. The predictor must learn the behavior of a branch in many different contexts independently. If the predictor has not seen a branch in a given context before, it can not make an informed prediction, even though it may have seen the same branch in many other contexts. This is a significant liability, but there are ways around it. Hybrid implementations, discussed later, address the problem by reverting back to a simpler predictor when the correlated predictor can not make an informed prediction.

A major issue in developing correlated predictors is the amount of information on which to correlate. In the case of local and global history predictors, this is the history length of

previous branch outcomes. Using more history increases the chances of including some previous information with which the branch behavior is correlated. But, using more history also increases the number of cases the predictor must learn. In the extreme case of infinitely long histories, each case would be unique and the predictor would never have any prior occurrences on which to base a prediction. Determining the best overall history length involves carefully balancing this tradeoff.

5.2.2 Implementing a predictor

Most work on predictors has focused on branch prediction. In branch prediction the predictor guesses if a conditional branch will be taken or not. Consider a global history branch predictor. Ideally the predictor would be able to keep information for every branch, based on every possible preceding dynamic path that led to the branch (considering dynamic paths of some fixed distance). This is obviously not practical as a predictor must be implemented with limited resources and must use algorithms that can be implemented with fast logic (one prediction per clock cycle).

There are a number of implementations of global history used for branch prediction based on a common approach described by McFarling [31] (see Figure 5-3). Like most branch predictors they are built around a table of 2-bit counters (that implement the functionality of encoding a prediction and one bit of hysteresis). They keep the outcomes of the previous *N* branches in a shift register and use this information along with the PC of the current branch instruction to index a table and make a prediction. The approaches use different hash functions to combine the PC and the global branch outcomes to produce a

fixed size index. The hash function usually consists of some variation of concatenating and combining (using an exclusive-or function) of the information, using a limited number of bits from either source.

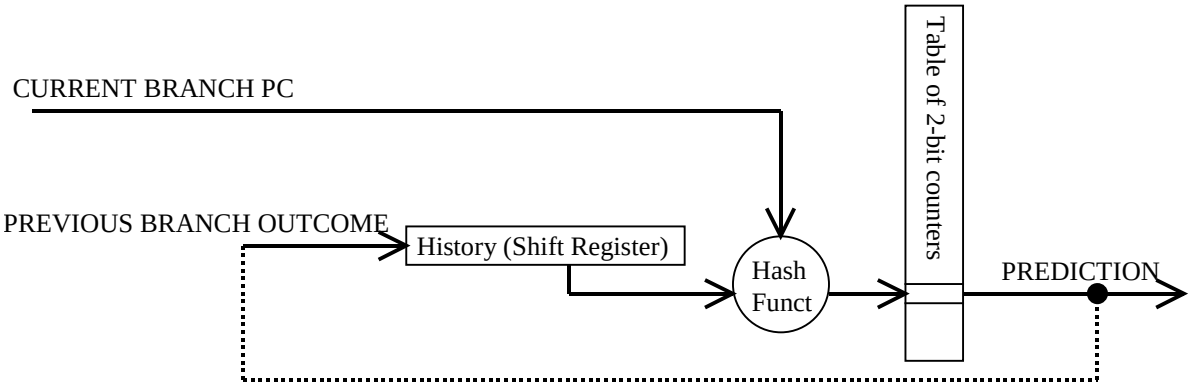


Figure 5-3 Example global history branch predictor.

The history of previous branch outcomes is a good approximation of the dynamic instruction sequence that led to the current branch. It is a good approximation in that different dynamic sequences tend to generate different history values. It is not sufficient to fully distinguish between all sequences. If two taken branches have the same branch history value at the time they are encountered and they have the same target, it is not possible to differentiate which path was taken from the perspective of the next branch. To be able to uniquely distinguish paths, it would be necessary to record at least some intermediate addresses along with, or instead of, branch outcomes. This approach is referred to as a *path-based* history scheme [36]. Even if all addresses were recorded in the history, the information would have to be compacted down to form an index, and some information would be lost. This aliasing or interference problem is a significant problem for all predictors.

5.2.2.1 Aliasing

Predictors are built around limited sized tables that store information of past behavior. These tables are usually in the range of 2^{10} to 2^{16} entries, which means they are indexed with a 10- to 16-bit index. The information used to identify the current branch and the history to correlate on is usually many more bits (it takes more bits to even uniquely identify a branch). The information is heavily compacted to form the index, and some information is lost. This leads to aliasing.

Aliasing occurs when two unrelated cases (branch and history combinations) reference the same entry in the predictor table, because of the hashing function used to form the index. This causes two problems. First, predictions are made for a current branch based on unrelated information on one or more other branches. Second, the current branch updates the information in the table, reducing the usefulness of the entry for other branch(es).

The more history that is used to correlate predictions, the more aliasing problems there will be. By using more history, each branch uses more entries in the table to differentiate between different contexts. The more total cases there are, the higher probability that any two (or more) will reference the same entry.

5.3 Implementing Next-Trace Prediction

I consider predictors designed specifically to work with trace caches. They predict traces explicitly, and in doing so, implicitly predict the control instructions within the trace. Next-trace predictors replace the conventional branch predictor, branch target buffer (BTB) [54]

and return address stack (RAS) [25]. They have low latency, and are capable of making a trace prediction every cycle. Next-trace predictors also offer accuracy comparable to the best conventional correlated branch predictors.

5.3.1 Correlated predictor

The core of the next-trace predictor uses correlation based on the history of the previous traces. The identifiers of the previous few traces represent a path history that is used to form an index into a prediction table (see Figure 5-4). Each entry in the table consists of the identifier of the predicted trace (PC and branch outcomes), and a 1-bit counter for hysteresis. When a prediction is correct the counter is set to one. When a prediction is incorrect and the counter is zero, the predicted trace will be replaced with the actual trace. Otherwise, the counter is set to zero.

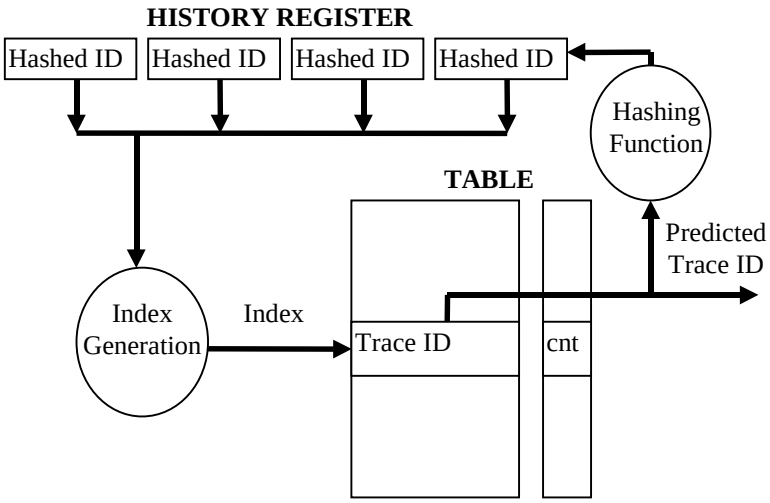


Figure 5-4 Correlated predictor.

Path history is maintained as a shift register that contains 16-bit hashed trace identifiers (Figure 5-4). The hashing function combines the trace starting address and branch outcomes (both encoded in the trace identifier) into a condensed encoding (see Figure 5-5). The

hashing function uses the outcome of the first two conditional branches in the trace identifier as the least significant two bits, the two least significant bits of the starting PC as the next two bits, the upper bits are formed by taking the outcomes of additional conditional branch outcomes and exclusive-oring them with the next least significant bits of the starting PC. Beyond the last conditional branch a value of zero is used for any remaining branch outcome bits.

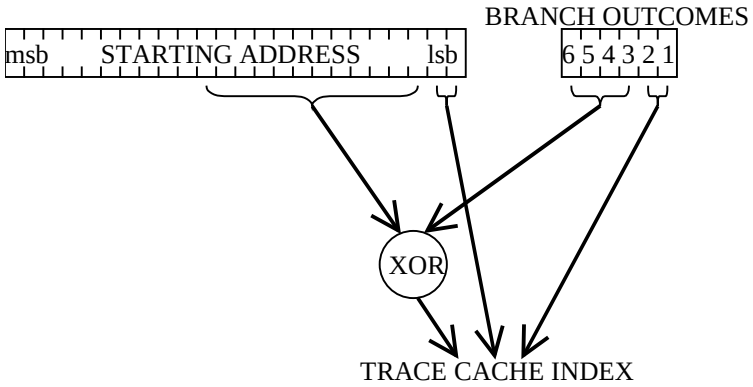


Figure 5-5 Hashing function.

The history register is updated speculatively with each new prediction. In the case of an incorrect prediction, the history is backed up to the state before the bad prediction. The prediction table is updated only after the last instruction of a trace is retired (it is not speculatively updated).

Ideally the index generation mechanism would simply concatenate the hashed identifiers from the history register to form the index. This is not practical because the prediction table is relatively small so the index must be restricted to a limited number of bits.

The index generation mechanism is based on the method developed by Bennett, Sharma, Smith and myself to do inter-task prediction for multiscalar processors [22]. The index

generation mechanism uses a few bits from each of the hashed trace identifiers to form an index. The low order bits of the hashed trace identifiers are used. More bits are used from more recent traces. The collection of selected bits from all the traces may be longer than the allowable index, in which case the collection of bits is folded over onto itself using an exclusive-or function to form the index. The "DOLC" naming convention [22] was developed for specifying the specific parameters of the index generation mechanism. The first variable, 'D'epth, is the number of traces besides the last trace that are used for forming the index. The other three variables are: number of bits from 'O'lder traces, number of bits from the 'L'ast trace and the number of bits from the 'C'urrent trace. In the example shown in Figure 5-6 the collection of bits from the trace identifiers is twice as long as the index, so it is folded in half and the two halves are combined with an exclusive-or. In other cases the bits may be folded into three parts, or may not need to be folded at all.

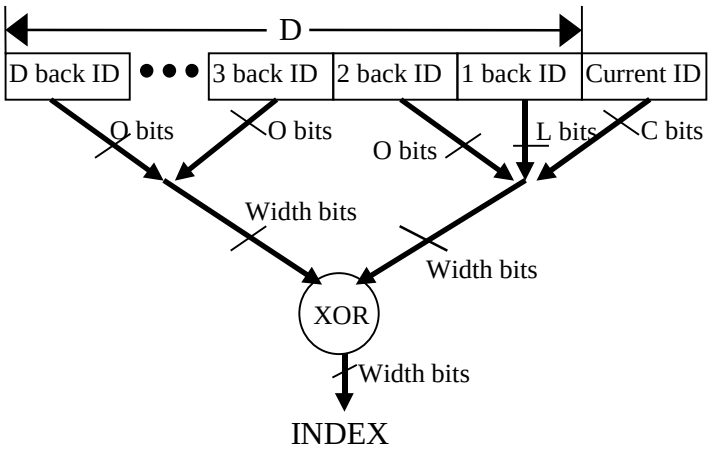


Figure 5-6 Index generation mechanism.

Choosing an appropriate DOLC configuration is a combination of "art and science." I have discovered a few heuristics through experimentation and experience. There are a couple important, but not immediately obvious, requirements of a good configuration. Values

should be chosen so that when the folding is done, none of the trace identifiers line up in the same bit position. That is, when the index is folded the least significant bit of one ID should never be combined with a non-least significant bit of another ID. Values should be chosen so that an ID is never folded with itself. There is always the tradeoff of wanting more bits from each ID and wanting to use as little folding as possible. Determining a good tradeoff point is basically trial and error directed by some intuitive feel. I will present some good configurations in the results sections which show trends in their organization.

5.3.2 Hybrid predictor

If the index into the prediction table addresses an entry that is unrelated to the current path history the prediction will almost certainly be incorrect. This can occur when the particular path has never occurred before, or because the table entry has been overwritten by unrelated path history due to aliasing. We have observed that both are significant, but for realistically sized tables aliasing is usually more important. In branch prediction, even a randomly selected table entry typically has about a 50% chance of being correct, but in the case of next-trace prediction the chances of being correct with a random table entry is very low.

To address this issue we operate a second, simpler predictor in parallel with the first (Figure 5-7). The secondary predictor requires a shorter learning time and suffers less aliasing pressure. The secondary predictor uses only the hashed identifier of the last trace to index its table. The prediction table entry is similar to the one for the correlated predictor

except a 4-bit saturating counter is used. The counter is incremented by one for correct predictions and decremented by 8 for incorrect predictions.

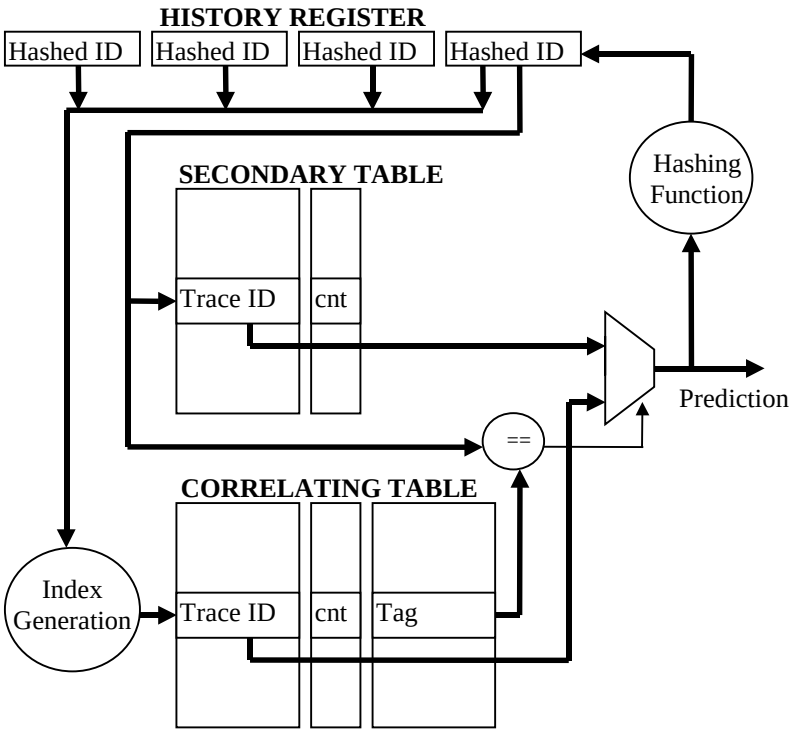


Figure 5-7 Hybrid predictor.

The larger counter is used to enable the detection of very consistent behavior. Identifying very consistent behavior is used for an optimization to reduce aliasing in the correlated predictor discussed at the end of this section. If the counter saturates, the same behavior must have been observed at least 15 times. Decrementing by 8 means that for the counter to saturate the same behavior must have been observed for at least 8 consecutive times. Decrementing by 8 also allows the predictor to adjust to changing behavior quickly, at least as quickly as a 2-bit hysteresis counter with normal increment and decrement.

To decide which predictor to use for any given prediction, a tag is added to the table entry in the correlated predictor. The tag is set with the low 10 bits of the hashed identifier of

the immediately preceding trace at the time the entry is updated. A 10-bit tag is sufficient to eliminate practically all unintended aliasing (the longer the tag the less likely undetected aliasing will occur, but the more space that is required by the predictor). When a prediction is being made, the tag is checked against the hashed identifier of the preceding trace, if they match, the correlated predictor is used; otherwise the secondary predictor is used. This method increases the likelihood that the correlated predictor corresponds to the correct context when it is used. This method also allows the secondary table to make a prediction when the context is very limited, i.e., under startup conditions.

The hybrid configuration enables the processor to tolerate aliasing pressure, and by modifying it slightly, the configuration can actually reduce aliasing pressure. If the 4-bit counter of the secondary predictor is saturated, its prediction is used, and more importantly, when it is correct the correlated predictor is not updated. This means if a trace is always followed by the same successor, the secondary predictor captures this behavior and the correlated predictor is not polluted. This reduces the number of updates to the correlated predictor and therefore the chances of aliasing. The relatively large counter, 4-bits, is used to avoid giving up the opportunity to use the correlated predictor unless there is high probability that a trace always has the same successor.

5.3.3 Return history stack (RHS)

The accuracy of the predictor is further increased by a new mechanism, the return history stack (RHS). A field is added to each trace indicating the number of calls it contains. If the trace ends in a return, the number of calls is decremented by one. After the path history is

updated, if there are any calls in the new trace, a copy of the most recent history is made for each call and these copies are pushed onto a special hardware stack. When there is a trace that ends in a return and contains no calls, the top of the stack is popped and is substituted for part of the history. The most recent entries from the current history within the subroutine are preserved, and the entries from the stack replace the remaining older entries of the history.

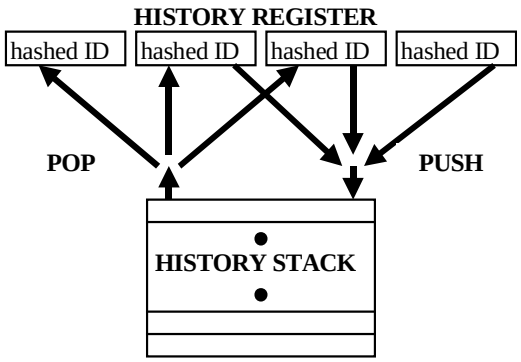


Figure 5-8 Return history stack implementation.

With the RHS, after a subroutine is called and has returned, the history contains information about what happened before the call, as well as knowledge of the last trace of the subroutine (see Figure 5-9). The RHS can significantly increase overall predictor accuracy. The reason for the increased accuracy is that control flow in a program after a subroutine is often tightly correlated to behavior before the call. Without the RHS the information before the call is often overwritten by the control flow within a subroutine. There is a tradeoff of how much information to use from before the call versus how much information to use from within the call. For different benchmarks the optimal point varies. I found that configurations using one entry from the subroutine provide consistently good behavior.

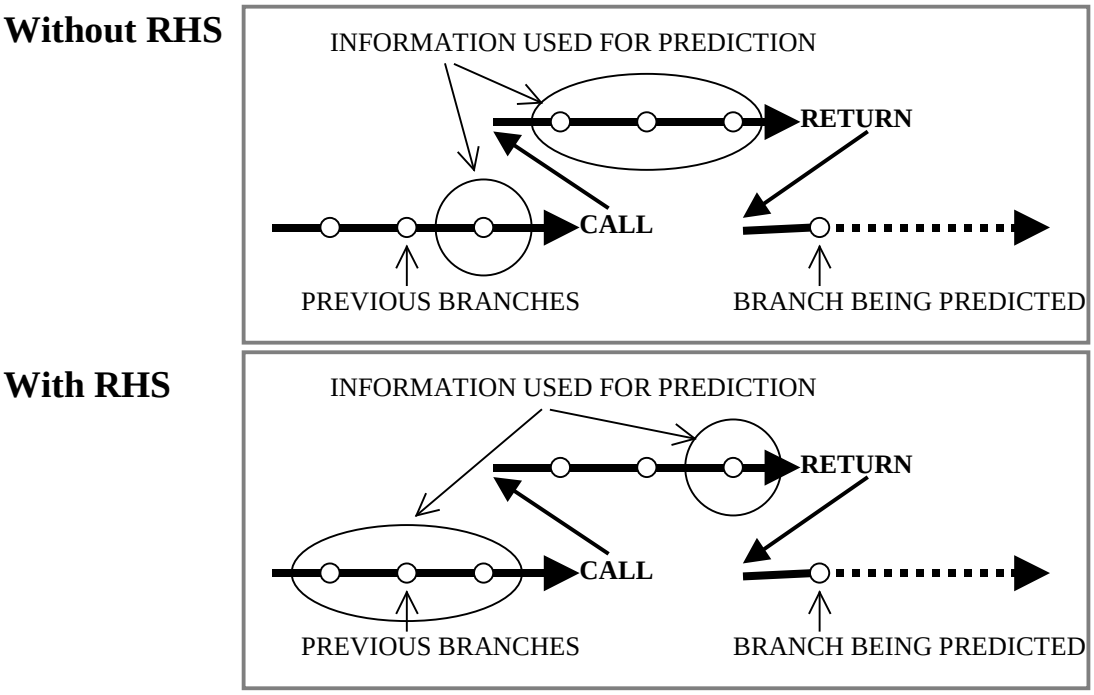


Figure 5-9 Examples of how a RHS works.

The next-trace predictor does not use a return address stack (RAS) [25], because it would require information on an instruction level granularity (individual target addresses), which the trace predictor is trying to avoid. The RHS can partly compensate for the absence of the RAS by helping in the initial prediction after a return. If a subroutine is significantly long it will force any pre-call information out of the history register. Hence, determining the calling routine, and therefore where to return, would be much harder without the RHS.

5.4 Performance with Unbounded Tables

This section presents quantitative results to demonstrate the potential performance of next-trace prediction. The SPECint95 benchmarks are used for the quantitative analysis. Corresponding results for the SPECfp95 benchmarks are given in Appendix A.

5.4.1 Performance potential of next-trace prediction without RHS

To determine the potential of next-trace prediction, an ideal implementation is considered. This is an implementation with unbounded storage and no aliasing, where each unique sequence of traces has its own entry to remember pervious occurrences. Each entry of the predictor consists of a prediction and a one-bit counter for hysteresis. Three different configurations of predictors are studied:

1. A predictor that always uses the ***maximum*** history length. If the exact sequence of traces has not been observed before, the predictor will have a compulsory miss.

2. A predictor that uses an ***optimal*** history length. It will match the longest history length it can, up to the maximum length, to provide a valid prediction. It only has compulsory misses after a new trace is observed.

3. A ***hybrid*** predictor that chooses between using the maximum history length or a history length of one. The maximum history length is used if it is valid and otherwise the short history length is used. This is a more realistic predictor to implement than the optimal history length, but provides much of the same benefit.

The prediction accuracy of the ideal predictors for length 16 traces is shown in Figure 5-10 through Figure 5-17. The graphs show the overall prediction accuracy, given in mispredictions per 1000 instructions for each of the three predictor configurations as a function of the maximum history length. The number of compulsory mispredictions observed for each configuration is also given (the optimal length and hybrid configurations have the same compulsory misprediction rate).

For all the benchmarks except *gcc*, *go* and *ijpeg*, the three predictor configurations perform comparably. For these benchmarks the number of compulsory mispredictions are insignificant. The prediction accuracy tends to increase (the number of mispredictions

decrease) for longer history lengths. This suggests that the path taken next is correlated with the path taken previously. The benefit of increasing the history length tends to decrease for longer history lengths.

For the benchmarks *gcc* and *go*, and to a lesser extent *ijpeg*, the number of compulsory mispredictions is significant for the predictor that always uses the maximum history length. The number of compulsory mispredictions grows substantially for longer history lengths, as there are more potential sequences. This causes the overall prediction accuracy of the maximum history length predictor to decrease (the number of mispredictions increase) after some history length. This demonstrates the tradeoff of correlation, where more information can be useful to make a prediction, but also leads to more unique cases to learn. The tradeoffs for correlation are even more pronounced when finite table sizes are used and aliasing becomes a significant issue.

The optimal history length configuration and the hybrid configuration enable the predictor to avoid most of the compulsory mispredictions for the benchmarks *gcc*, *go* and *ijpeg*. The benefit of these configurations becomes more important for longer maximum history lengths. By using these configurations, the overall prediction accuracy continues to go down as the maximum history length is increased. The hybrid configuration is able to perform nearly as well as the optimal history length configuration, although some deviation is seen for long maximum history lengths.

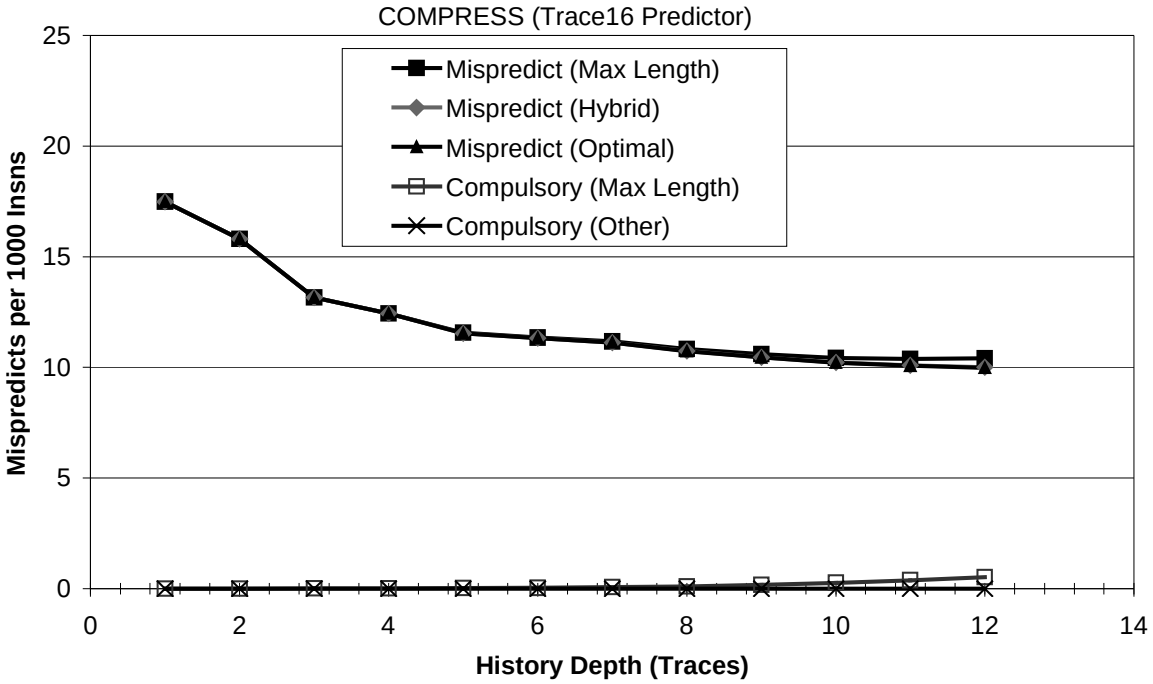


Figure 5-10 Prediction accuracy without RHS for *compress* with length 16 traces.

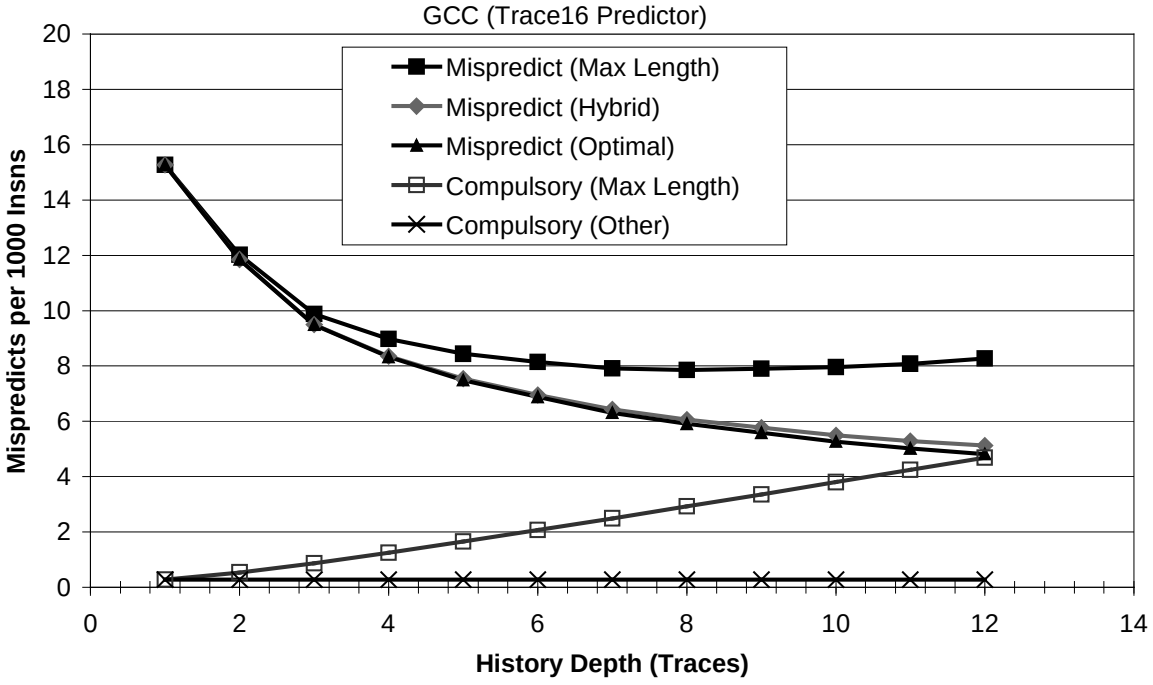


Figure 5-11 Prediction accuracy without RHS for *gcc* with length 16 traces.

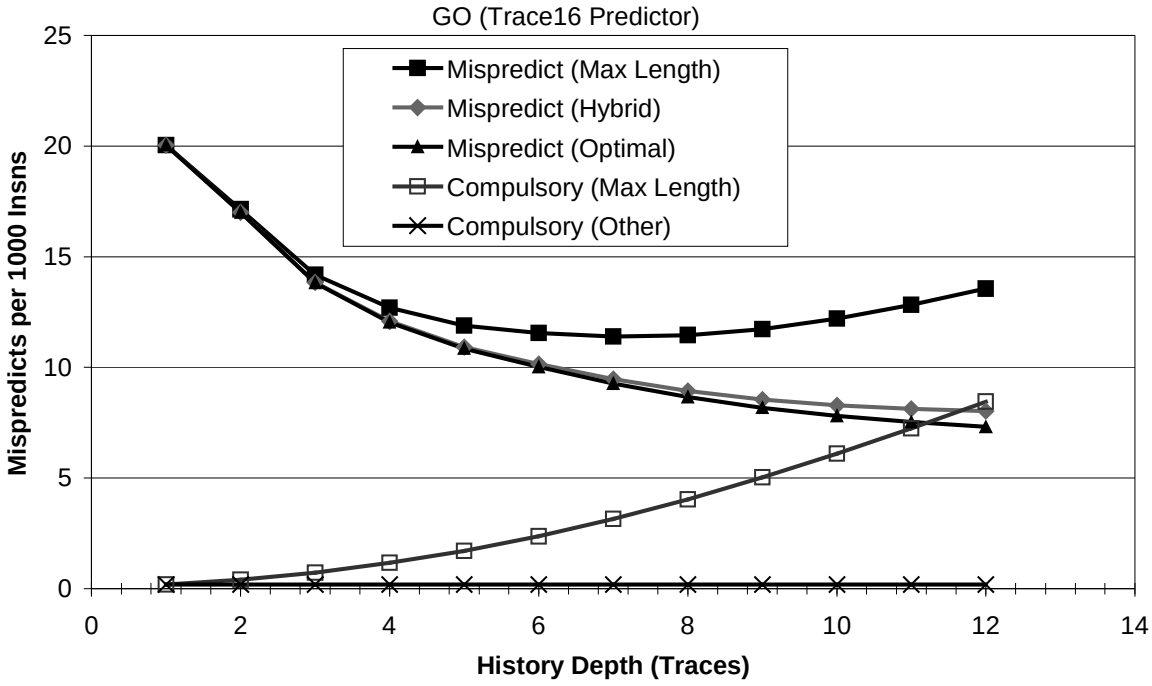


Figure 5-12 Prediction accuracy without RHS for *go* with length 16 traces.

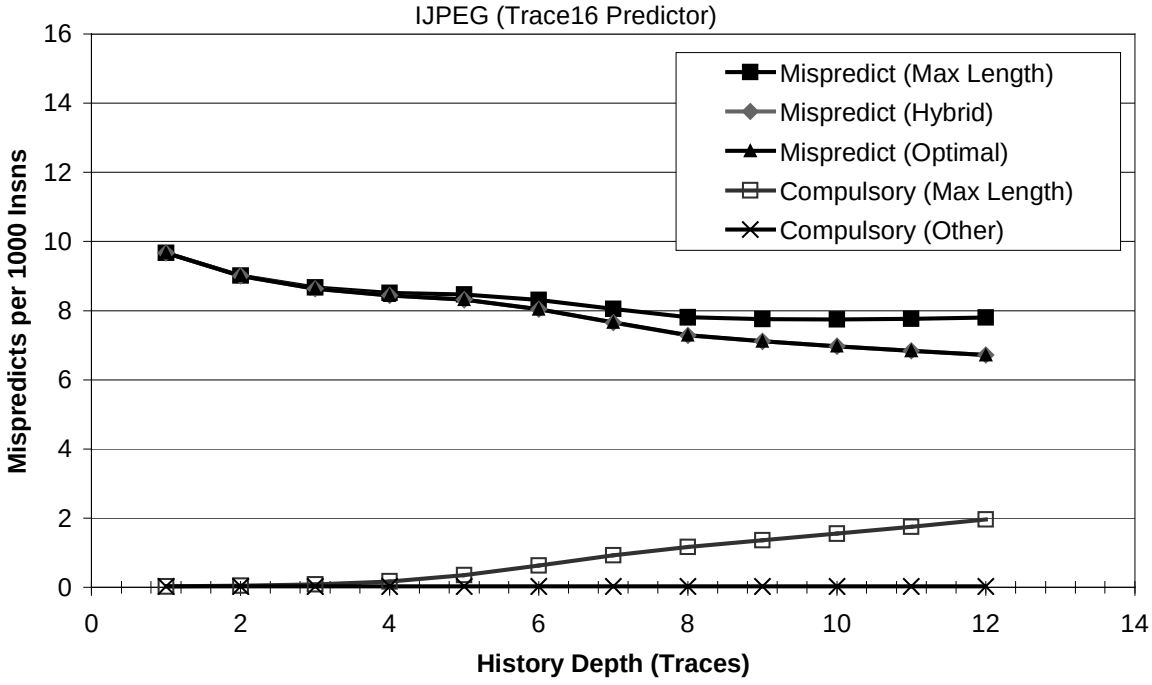


Figure 5-13 Prediction accuracy without RHS for *ijpeg* with length 16 traces.

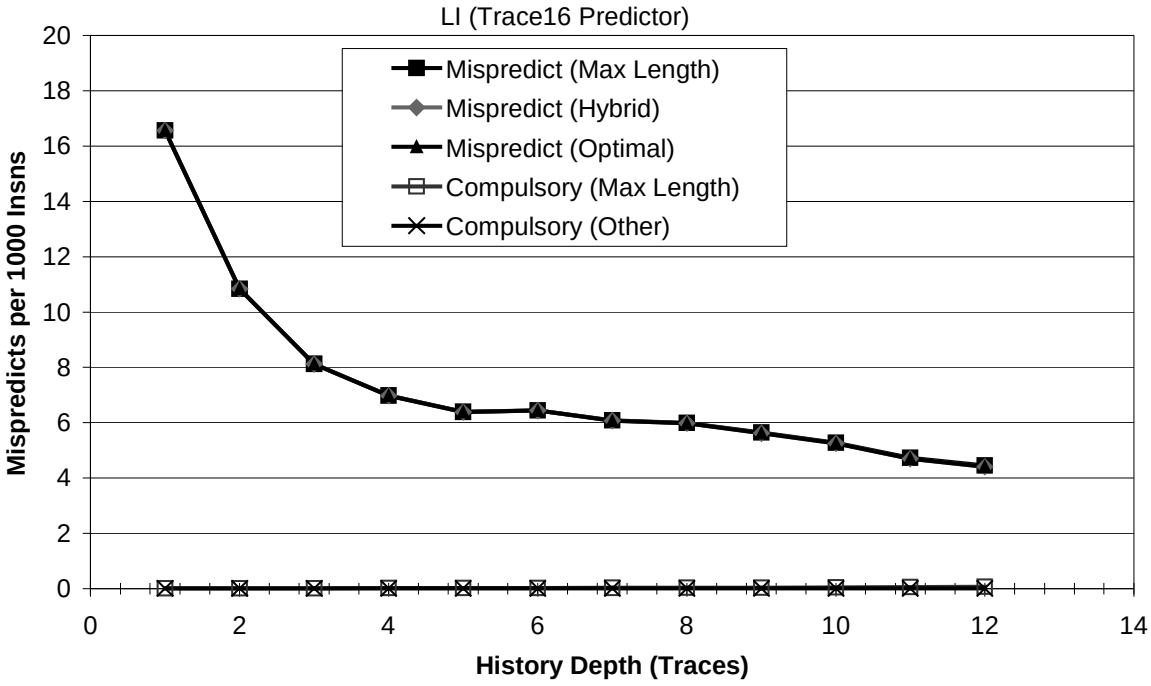


Figure 5-14 Prediction accuracy without RHS for *li* with length 16 traces.

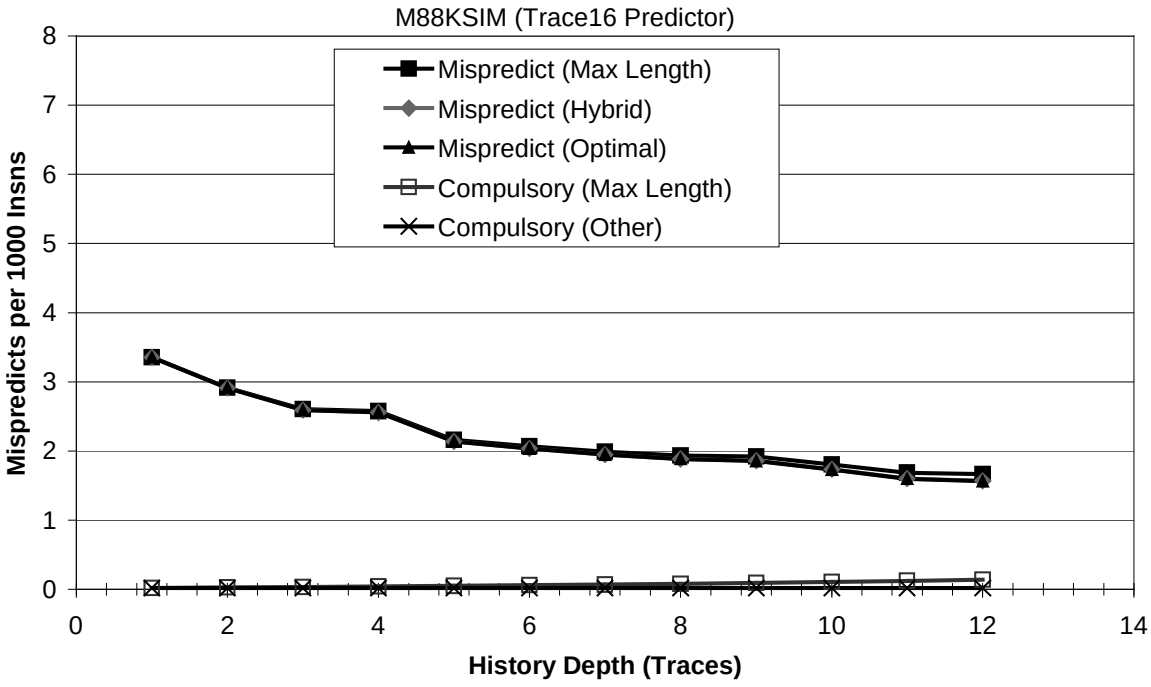


Figure 5-15 Prediction accuracy without RHS for *m88ksim* with length 16 traces.

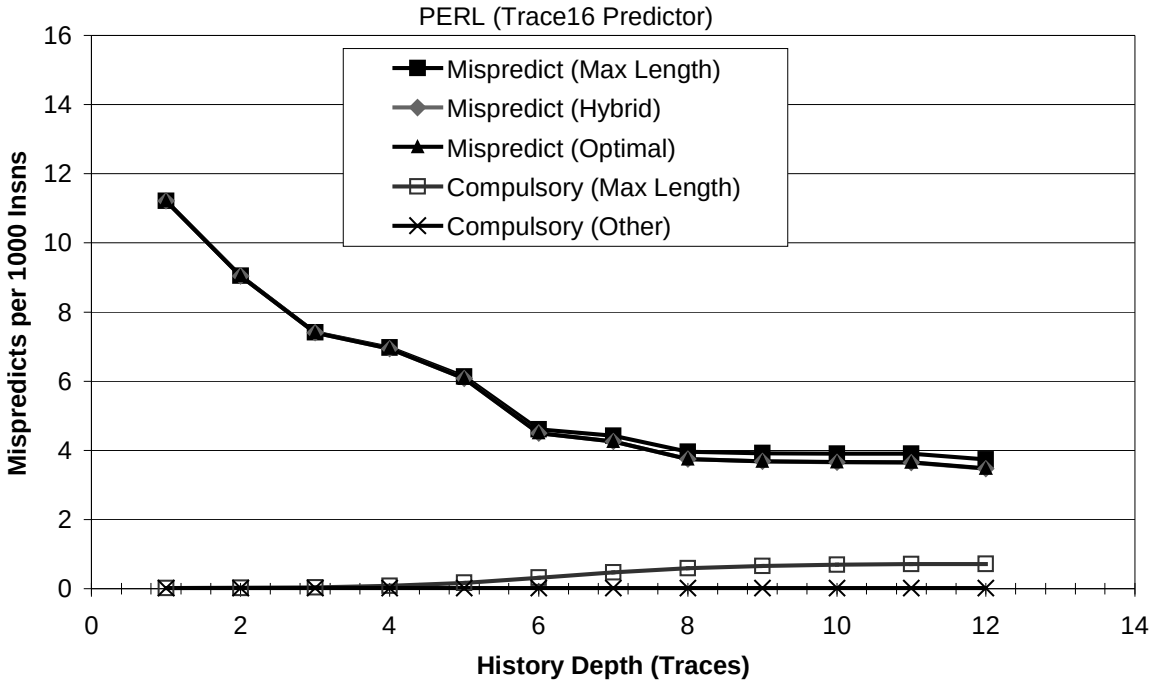


Figure 5-16 Prediction accuracy without RHS for *perl* with length 16 traces.

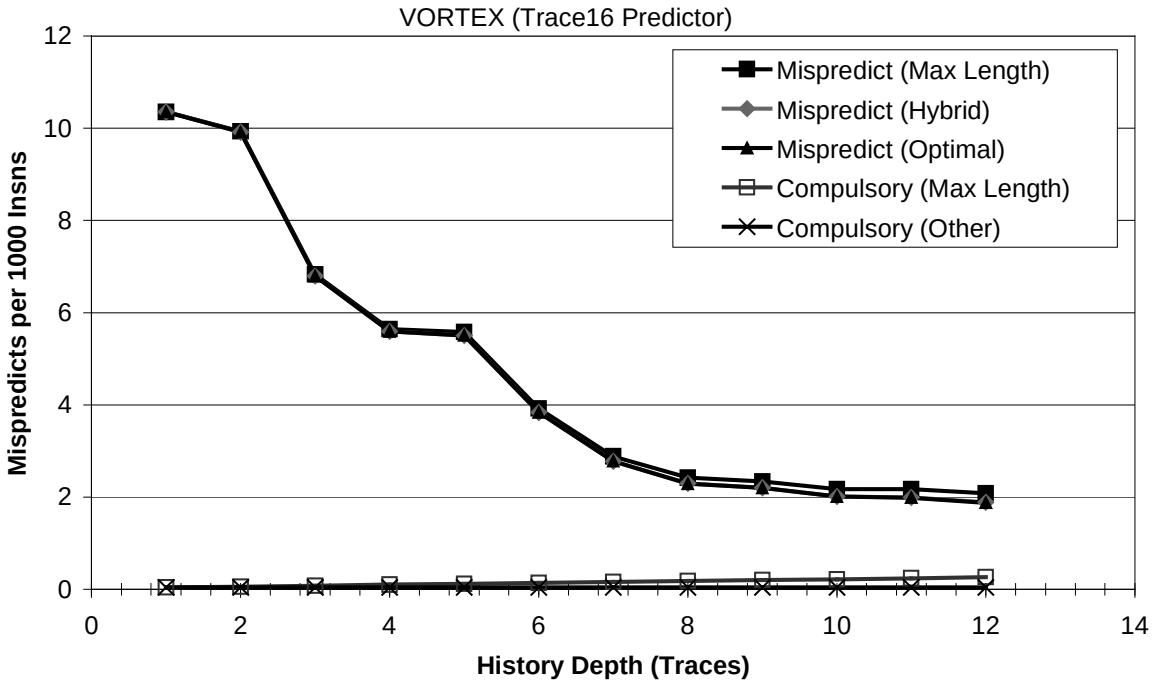


Figure 5-17 Prediction accuracy without RHS for *vortex* with length 16 traces.

Similar behavior is seen for length 32 traces as compared with length 16 traces (refer to Figure 5-18 through Figure 5-25). Length 32 traces are on average longer than length 16 traces, however they are approximately 50% longer rather than twice as long due to the trace selection heuristics. With length 32 traces there is more path information encoded in each trace, so there is more information in a history with the same number of traces as compared with length 16 traces. For very long histories, some of the benchmarks that did not observe significant compulsory mispredictions for length 16 traces see an increase for length 32 traces.

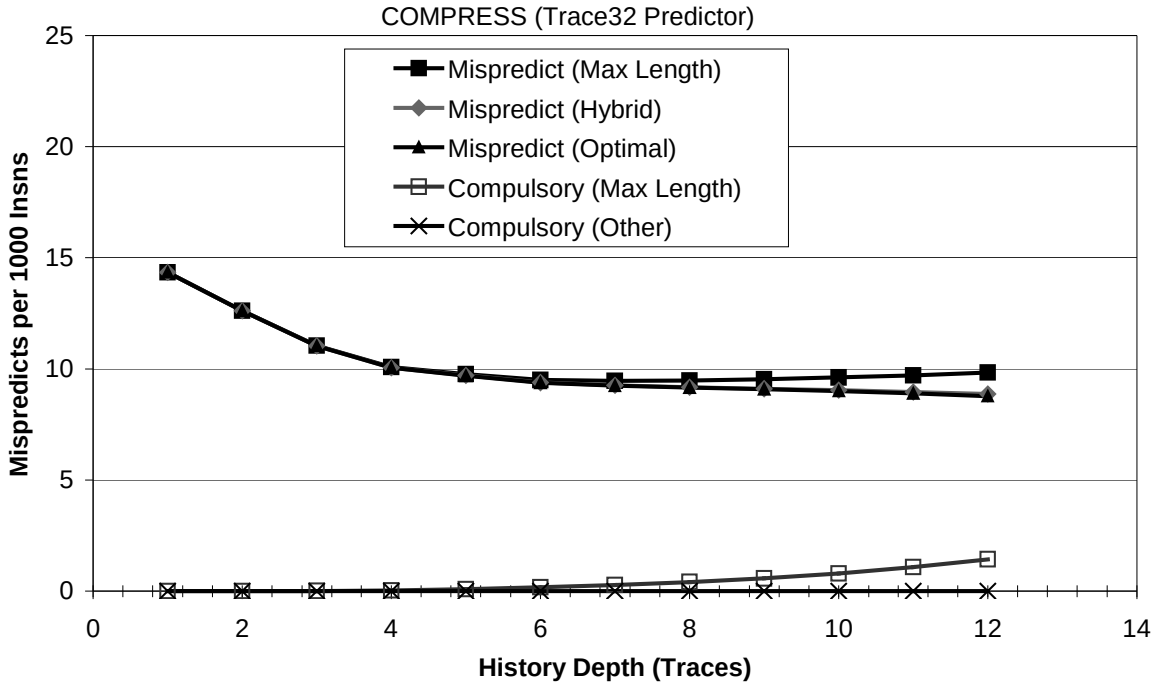


Figure 5-18 Prediction accuracy without RHS for *compress* with length 32 traces.

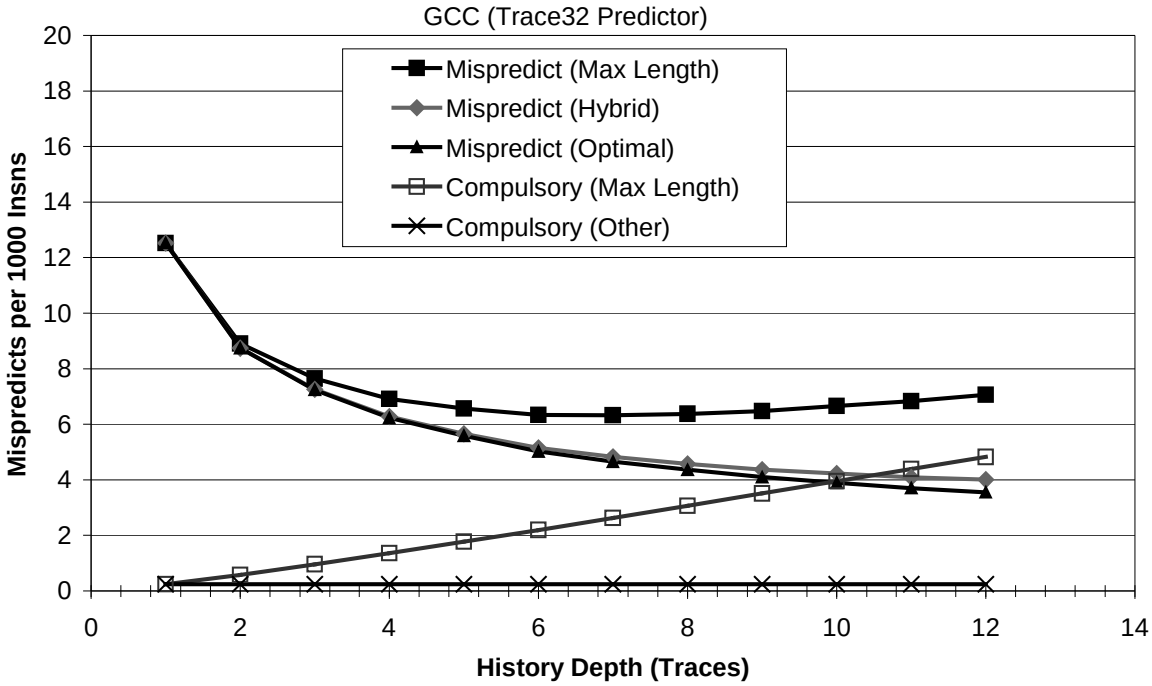


Figure 5-19 Prediction accuracy without RHS for *gcc* with length 32 traces.

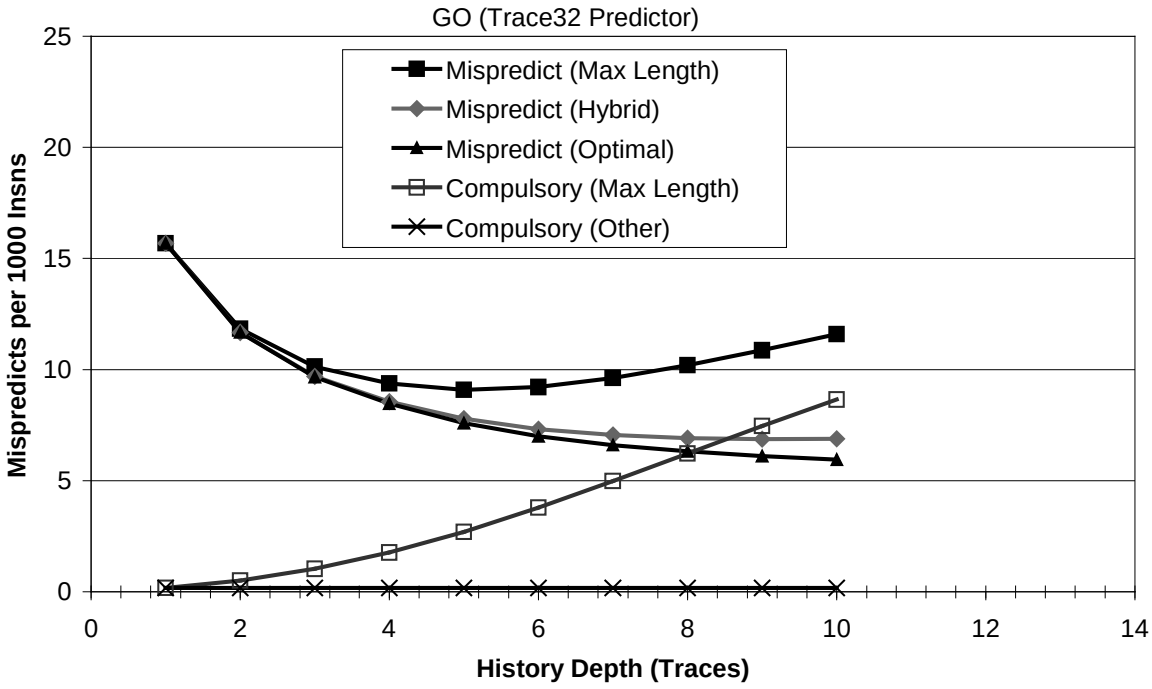


Figure 5-20 Prediction accuracy without RHS for *go* with length 32 traces.

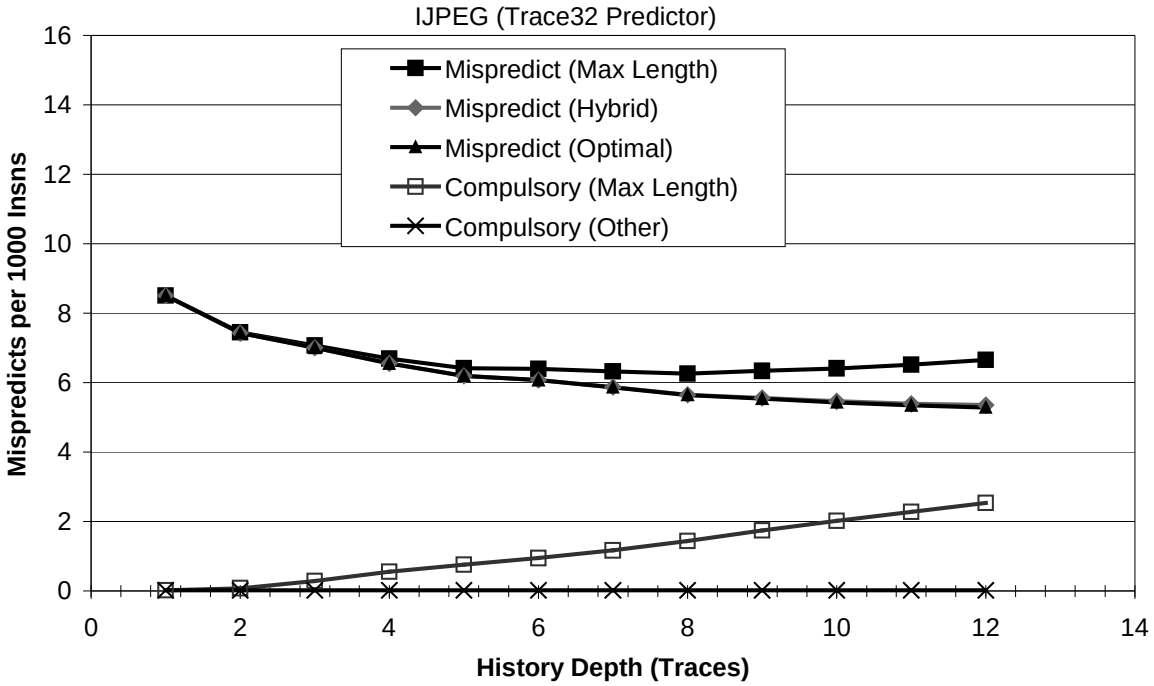


Figure 5-21 Prediction accuracy without RHS for *ijpeg* with length 32 traces.

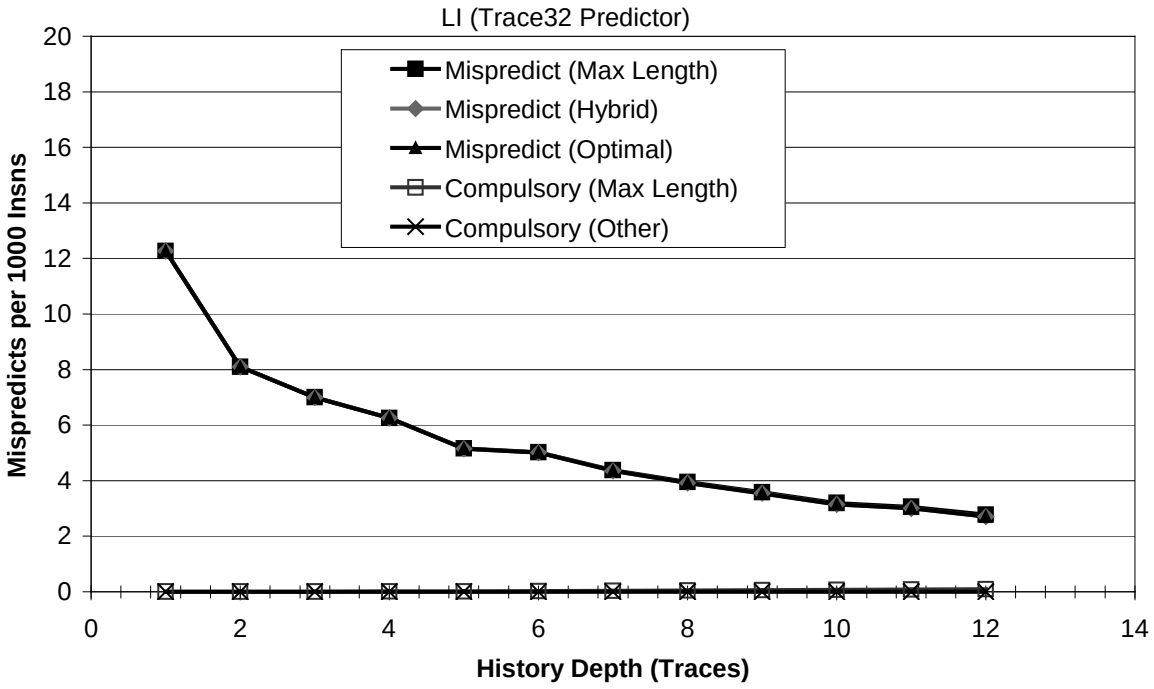


Figure 5-22 Prediction accuracy without RHS for *li* with length 32 traces.

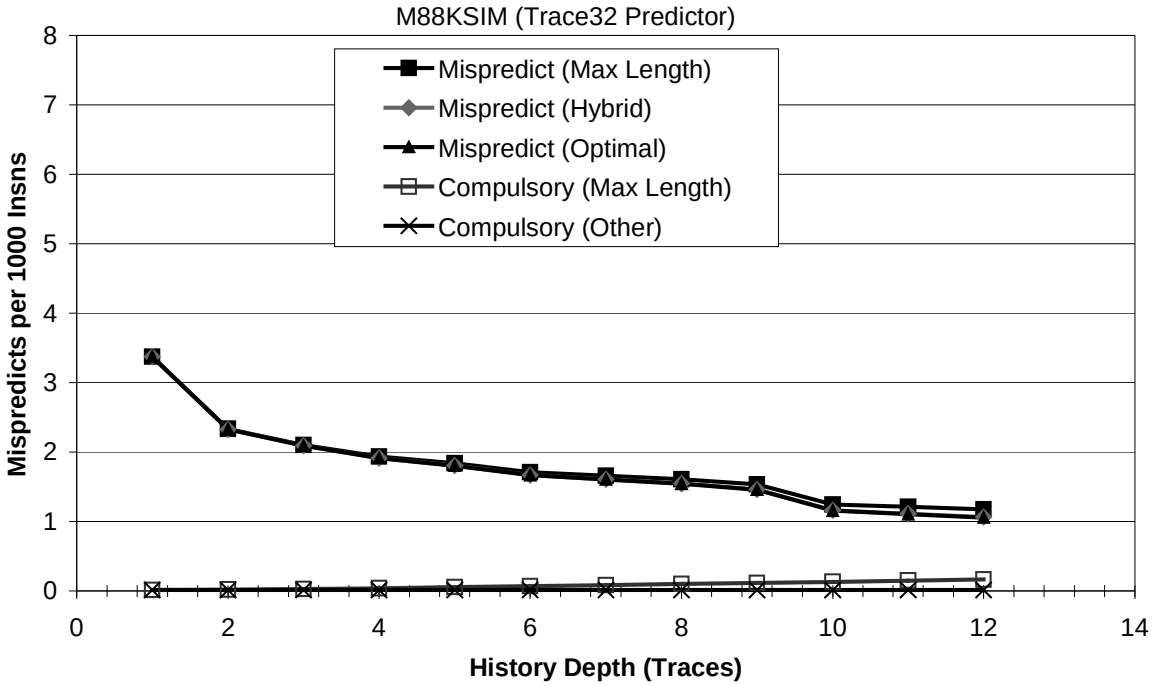


Figure 5-23 Prediction accuracy without RHS for *m88ksim* with length 32 traces.

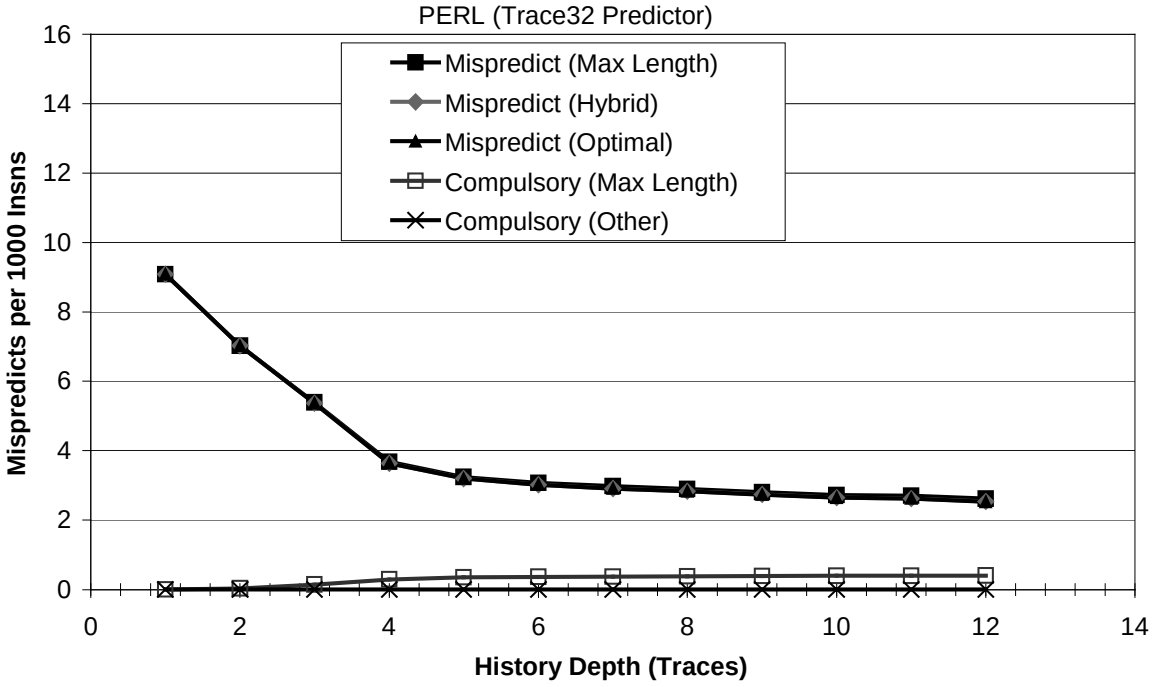


Figure 5-24 Prediction accuracy without RHS for *perl* with length 32 traces.

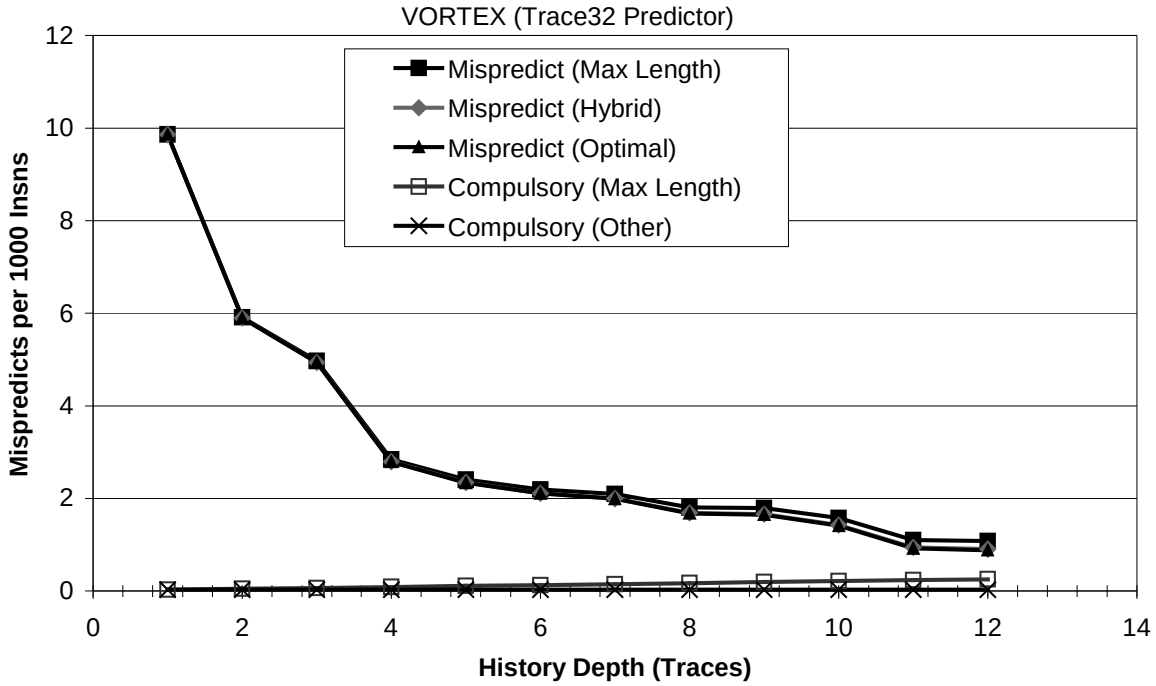


Figure 5-25 Prediction accuracy without RHS for *vortex* with length 32 traces.

5.4.2 Performance potential of next-trace prediction with RHS

A return history stack (RHS) can increase the performance of a correlated predictor based on global history if conditional branches after a subroutine are better correlated on history before the subroutine call than history within the subroutine. Individual branches within an application behave differently and are correlated on different information and the distribution of branches is different between applications. But, in general, a RHS increases prediction accuracy.

Figure 5-26 through Figure 5-33 show the performance of different RHS configurations for each of the benchmarks for length 16 traces. The configurations correspond to how much history from the subroutine (in terms of the number of traces) is preserved; the rest of the information from the subroutine is discarded so that more history from before the subroutine

call can be used. The predictor used is a hybrid predictor (one correlated predictor and one simple predictor) with the RHS applied to the history of the correlated predictor. The results for length 32 traces are very similar and were therefore not included.

Most of the benchmarks see a notable benefit from the RHS. *vortex* in particular sees a very substantial reduction in mispredictions, up to 90%, from the RHS. Other benchmarks see smaller but still significant reduction in mispredictions, with an overall average reduction in mispredict rate of 13% (with a harmonic mean of 4%). The benchmarks *compress* and *li* experience very small increases in mispredictions at longer history depths from the RHS. In general a RHS that preserves only the last trace from the subroutine (RHS 1) performs as well or better than other RHS configurations.

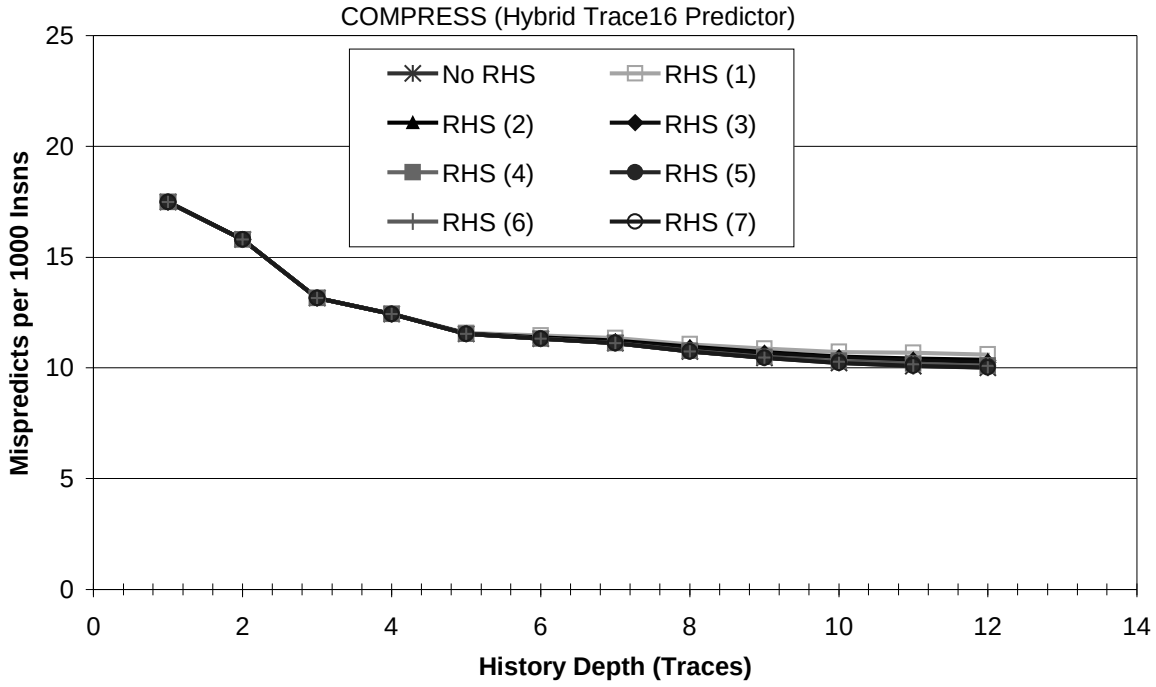


Figure 5-26 Effect of adding a RHS for *compress*.

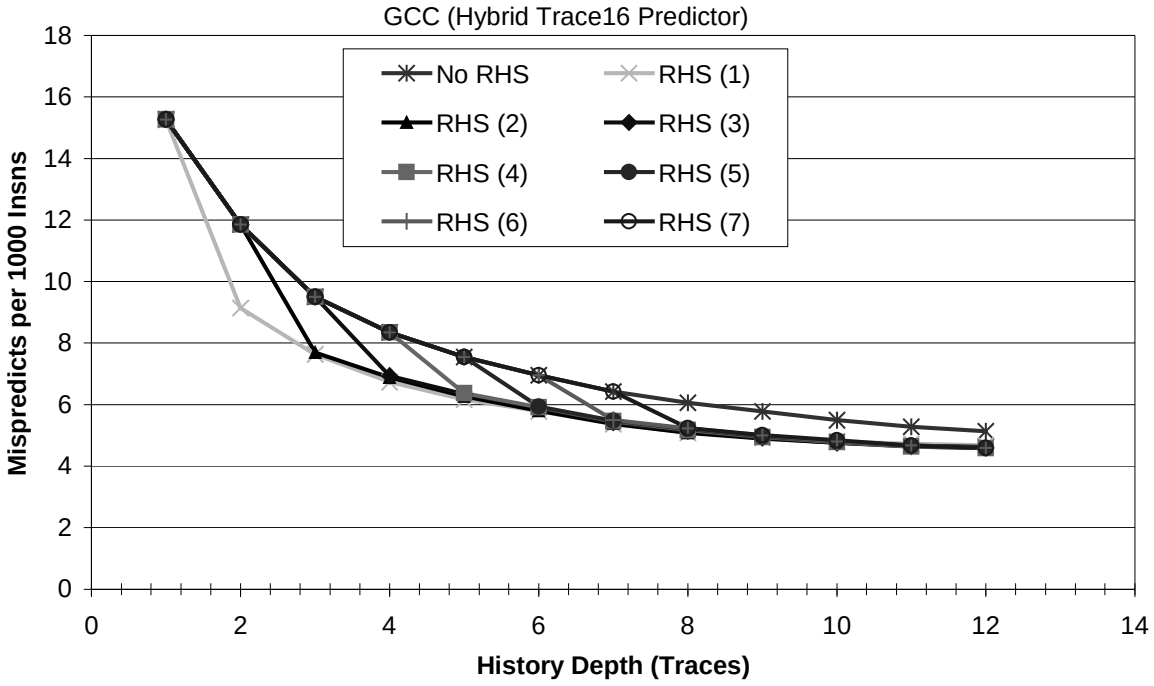


Figure 5-27 Effect of adding a RHS for *gcc*.

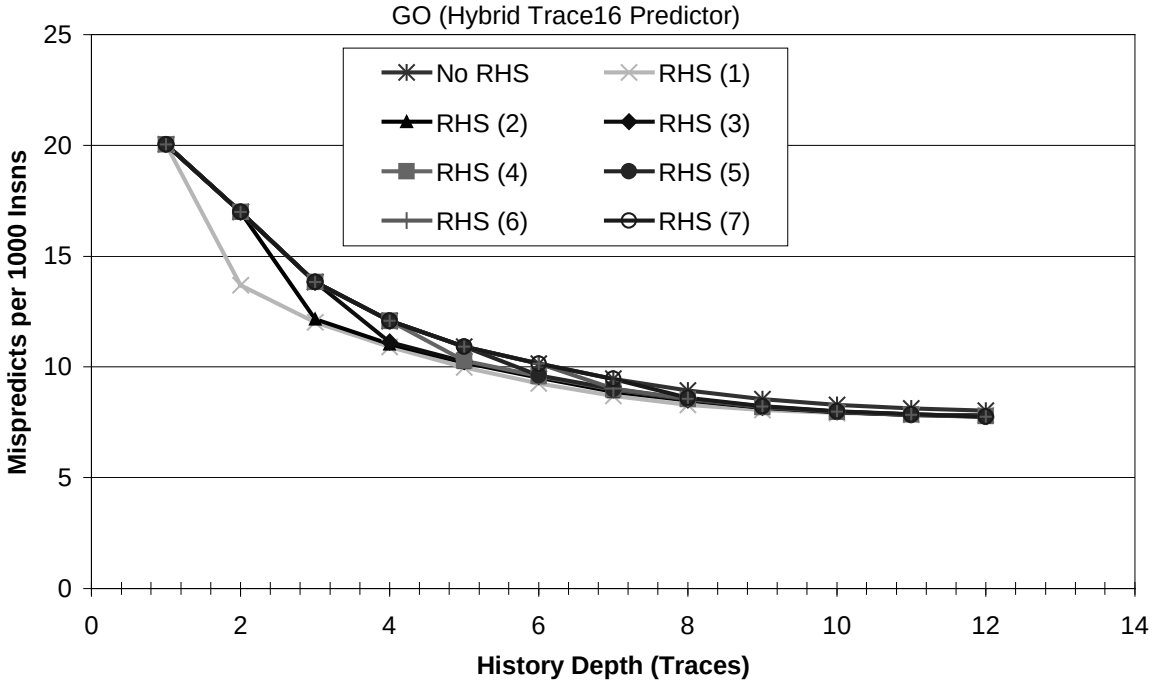


Figure 5-28 Effect of adding a RHS for *go*.

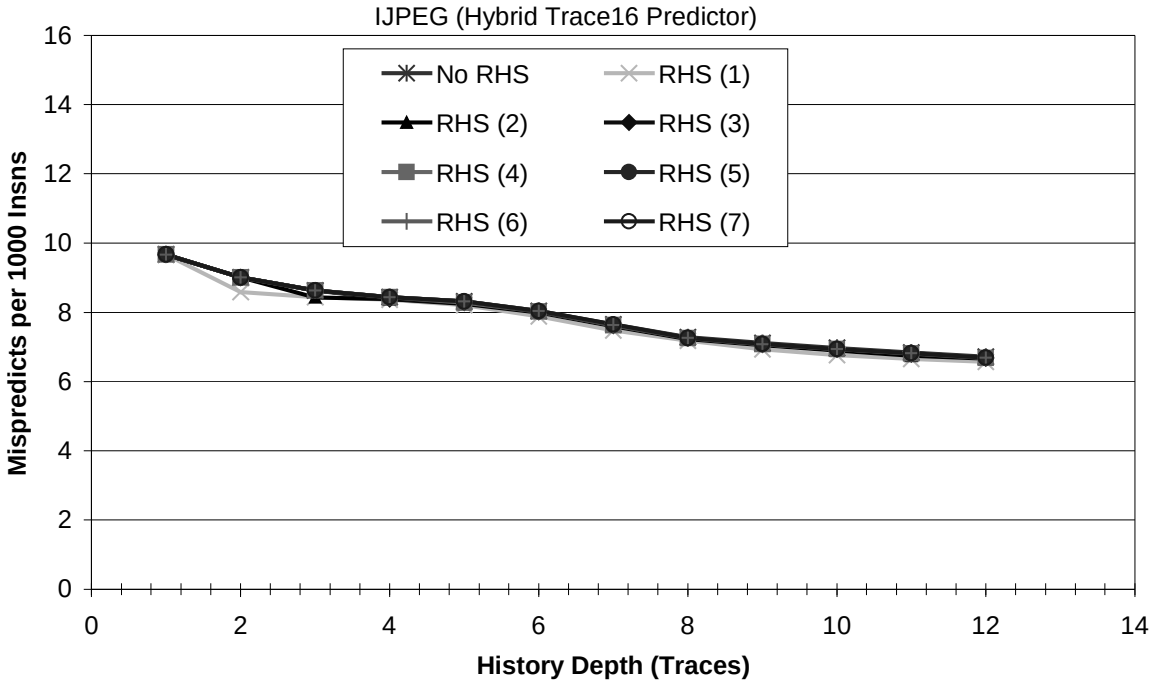


Figure 5-29 Effect of adding a RHS for *ijpeg*.

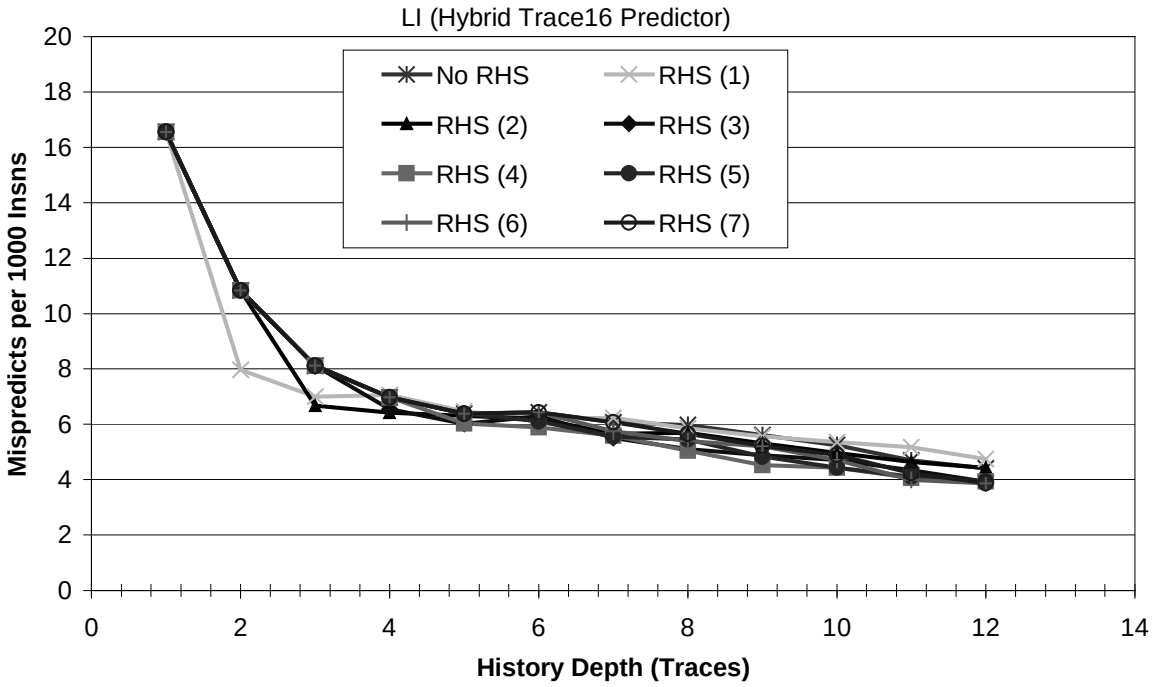


Figure 5-30 Effect of adding a RHS for *li*.

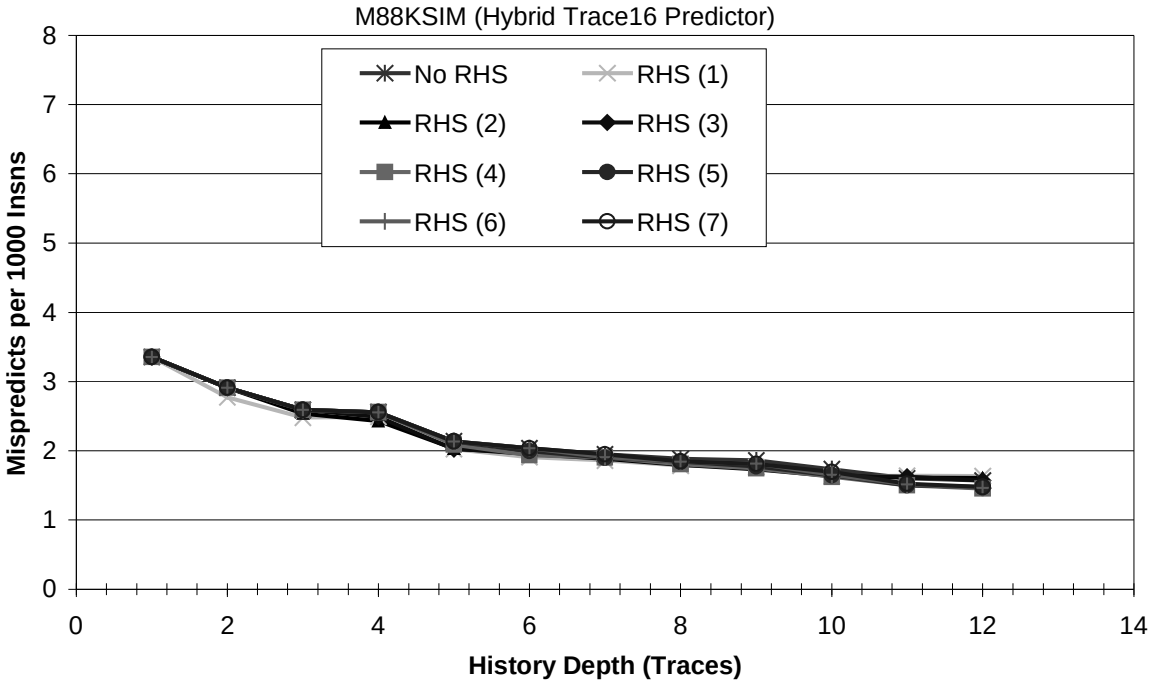


Figure 5-31 Effect of adding a RHS for *m88ksim*.

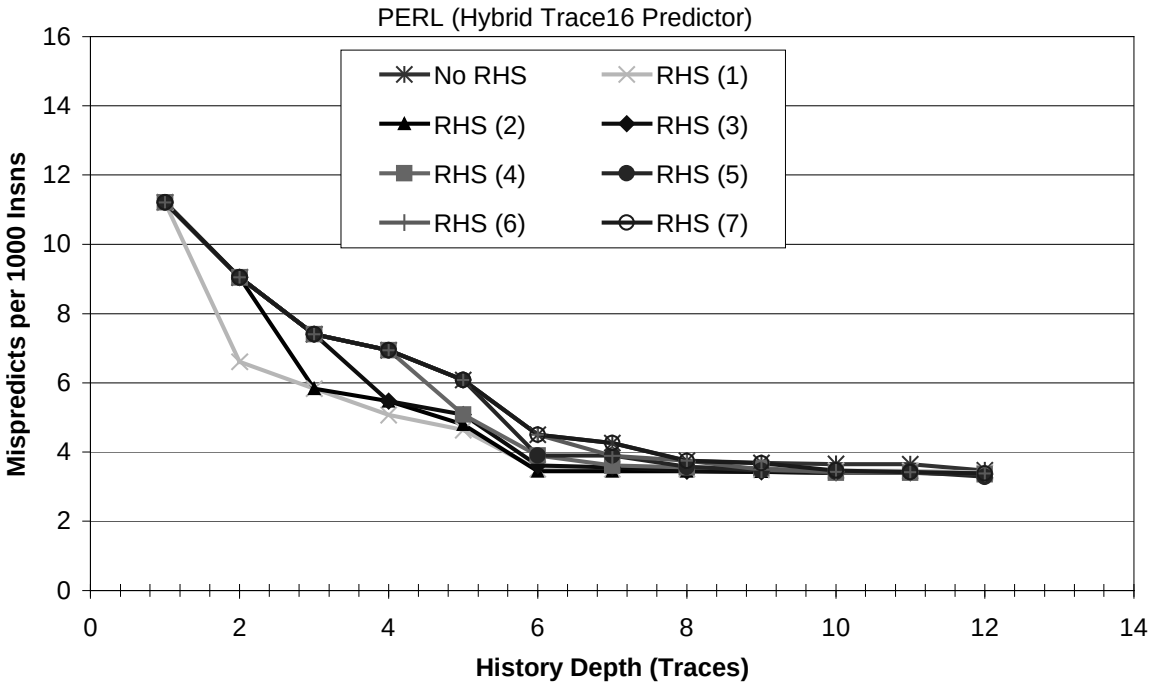


Figure 5-32 Effect of adding a RHS for *perl*.

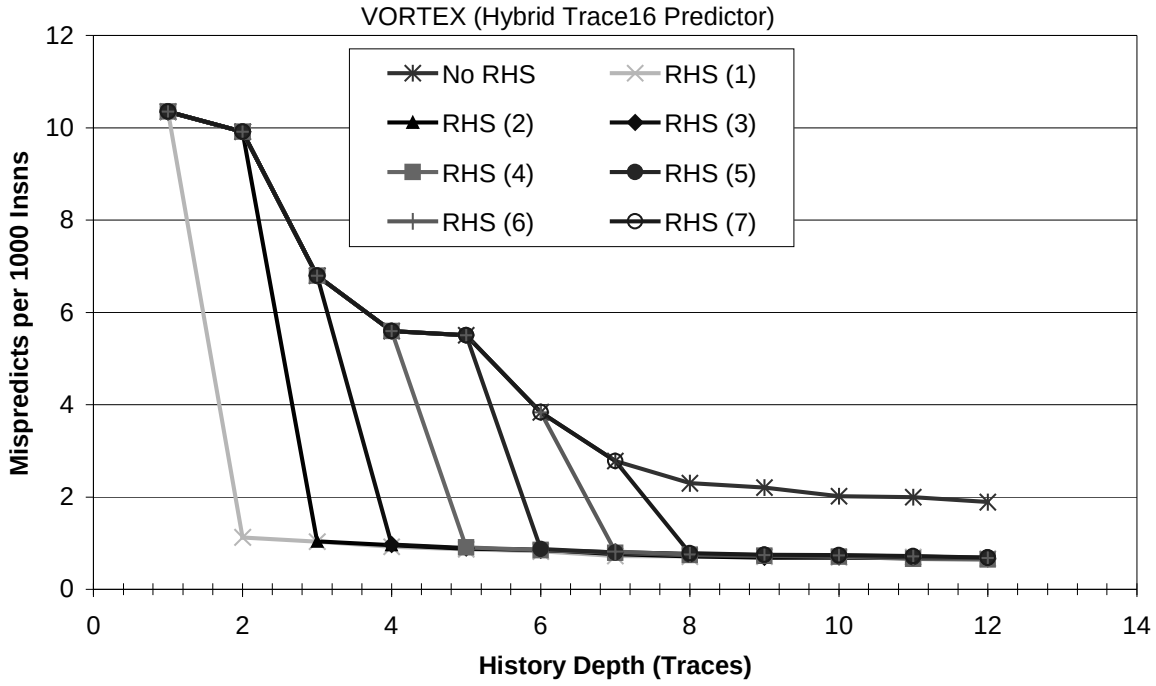


Figure 5-33 Effect of adding a RHS for *vortex*.

The RHS can improve performance for any global history predictor. The performance improvement is more substantial for the next-trace predictor because it helps compensate for the absence of an RAS (see Section 5.3.3). Even for a conventional branch predictor, the RHS increases the prediction accuracy notably. Figure 5-34 and Figure 5-35 show the benefit of adding a RHS to an ideal (no aliasing) correlated branch predictor for the benchmarks *gcc* and *vortex* respectively (two of the benchmarks that see the most benefit from a RHS). The behavior for the branch predictor is similar to the behavior of the next-trace predictor, with the difference being the magnitude of the penalty for not using a RHS.

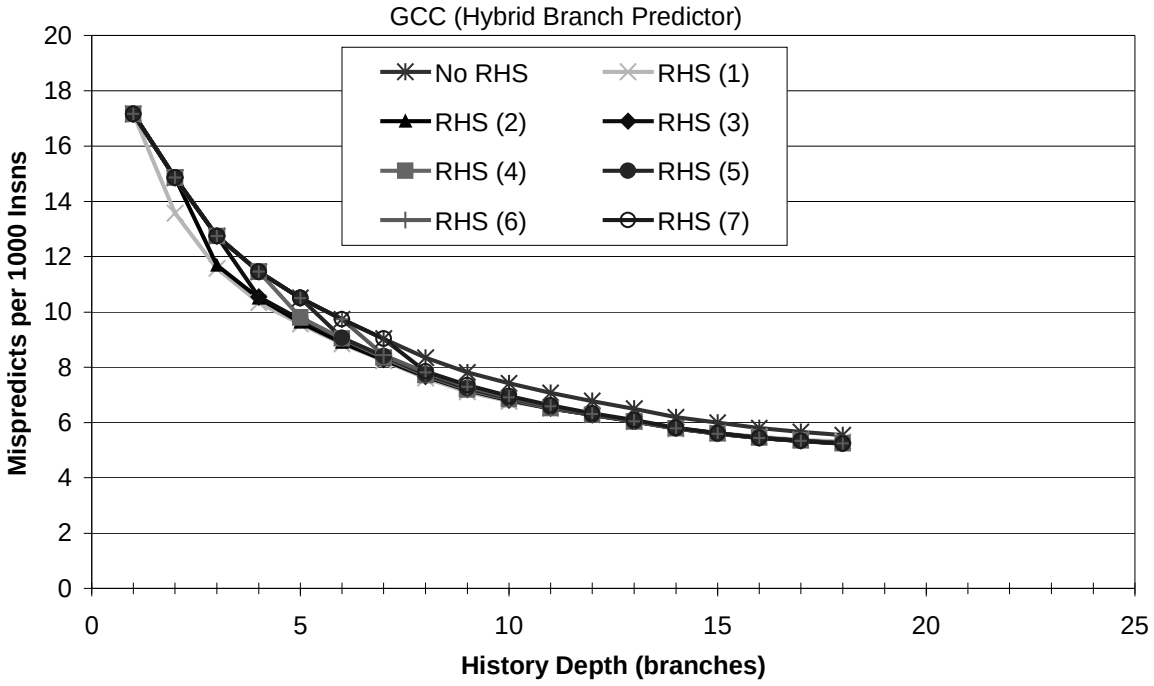


Figure 5-34 Branch prediction with a RHS for *gcc*.

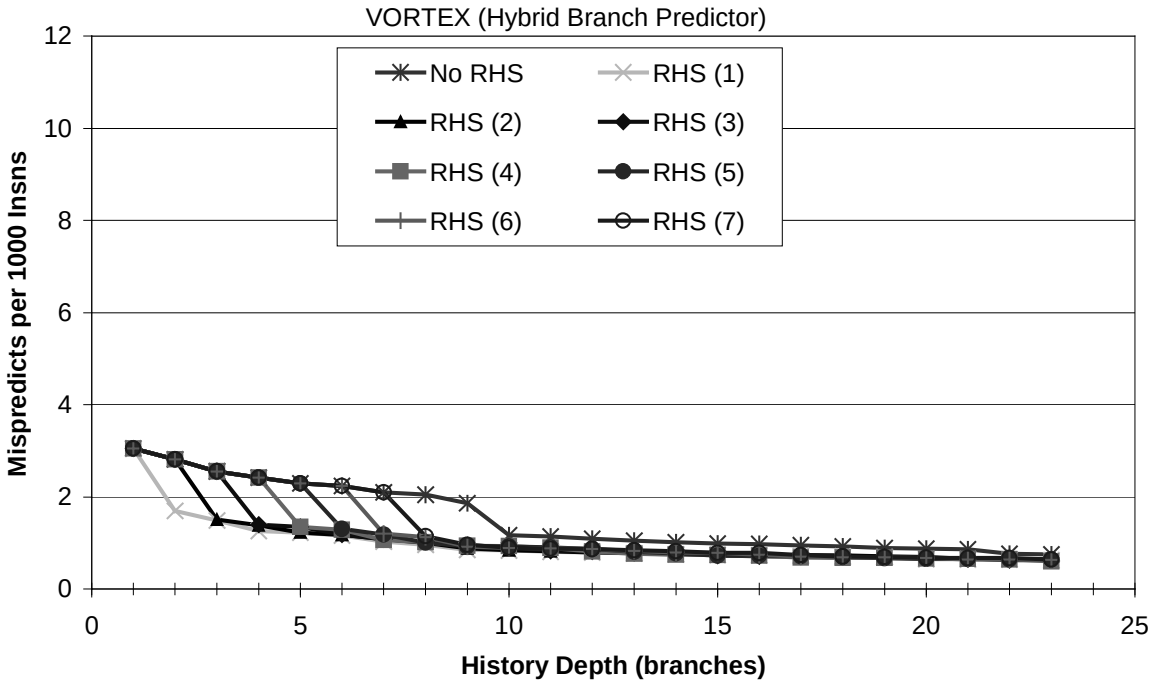


Figure 5-35 Branch prediction with a RHS for *vortex*.

5.4.3 Comparison of next-trace prediction with branch prediction

An important issue is how next-trace prediction compares to multiple-branch prediction. In the previous section I demonstrated the performance potential of the next-trace predictor. But, how does it compare with what a more traditional branch predictor can achieve? In this section I address this issue and discuss why the performance differs.

To avoid having implementation decisions overshadow the fundamental differences, ideal implementations are considered for the next-trace predictor and the branch predictor. The next-trace predictor is the hybrid (a correlated predictor and a simple predictor) discussed in the previous section. The branch predictor is a similar predictor that uses a history of the previous few conditional branches, the address and outcome of the branch, along with the address of the current branch to reference a unique entry that consists of a 2-bit counter. This is the most ideal branch predictor that can be constructed based on global history. The branch predictor is also a hybrid configuration that uses a correlated predictor and a simple predictor that is used when the correlated predictor does not have a valid entry. Both the next-trace predictor and the branch predictor have a RHS for the correlated predictor. The results for the best RHS configuration are given for each history depth.

The branch predictor is assumed to have a perfect branch target predictor, even for the targets of indirect branches and returns, which is optimistic. The next-trace predictor implements the true target prediction as part of its operation.

A comparison of a branch predictor, a next-trace predictor for length 16 traces and a next-trace predictor for length 32 traces is given in Figure 5-36 through Figure 5-43. The

results are presented in mispredictions per 1000 instructions as a function of the average history length. The average history length is the number of branches in the history multiplied by the average distance between branches, or the number of traces in the history multiplied by the average trace length.

The most important thing to observe from the results is that the performance is very comparable for branch prediction and next-trace prediction. Next-trace prediction offers the advantages of providing the bandwidth of control predictions needed for the trace cache with a low latency. If there are no significant penalties for using it, as compared with branch prediction, these benefits make it attractive to use with a trace cache.

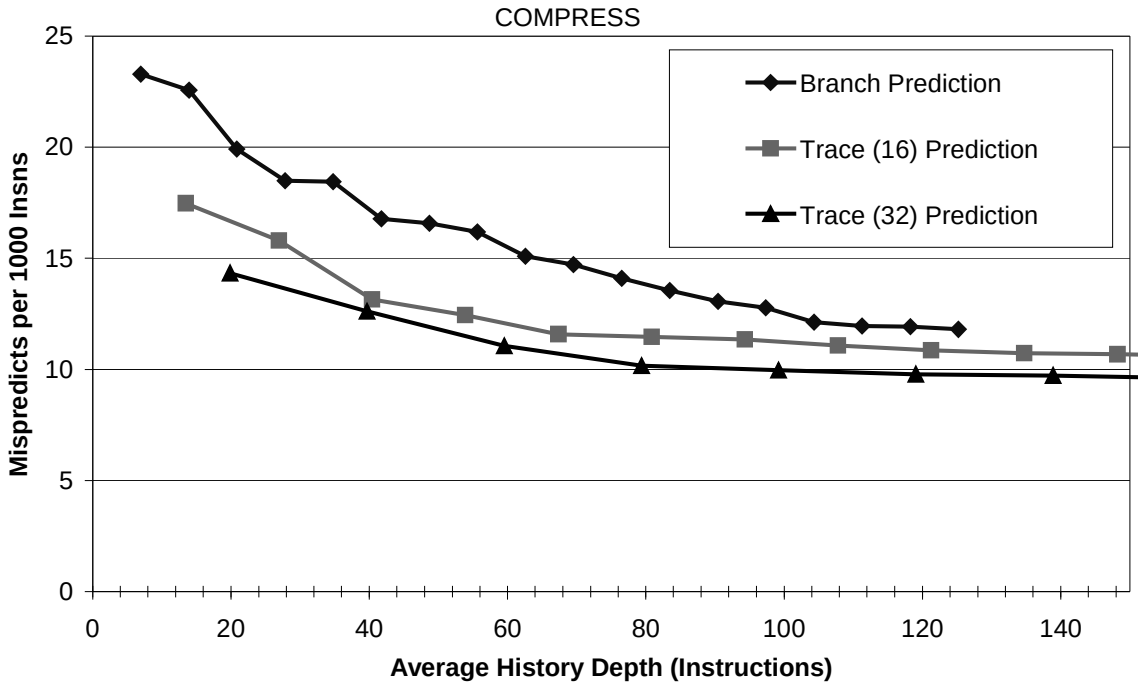


Figure 5-36 Comparisons of predictors for *compress*.

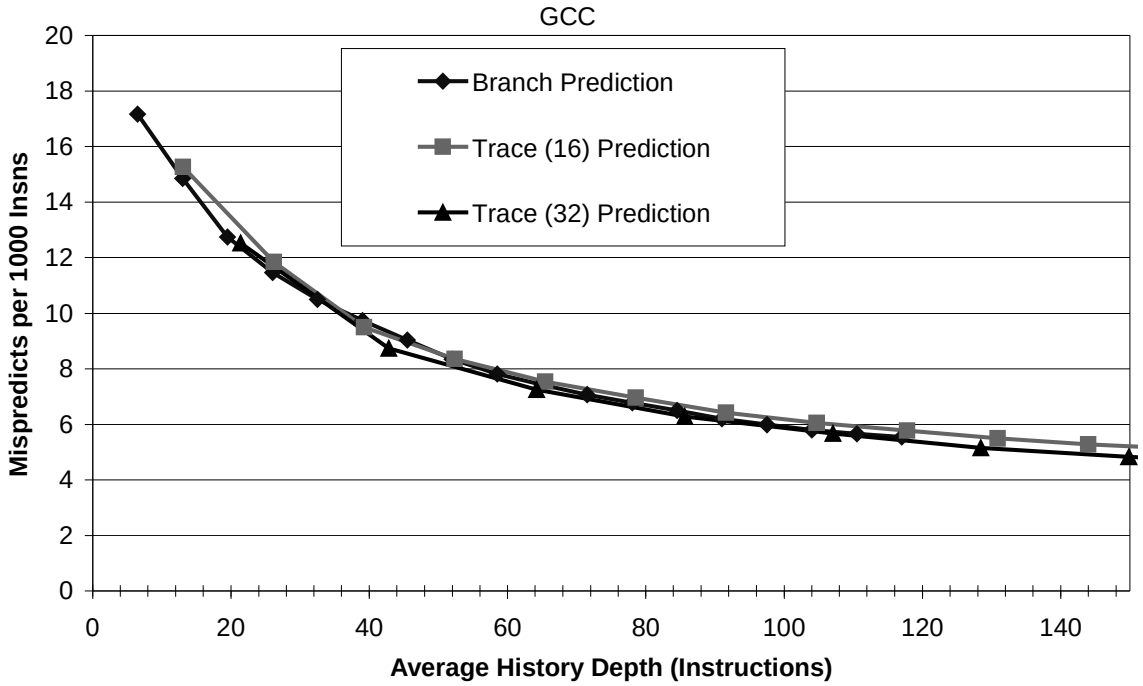


Figure 5-37 Comparisons of predictors for *gcc*.

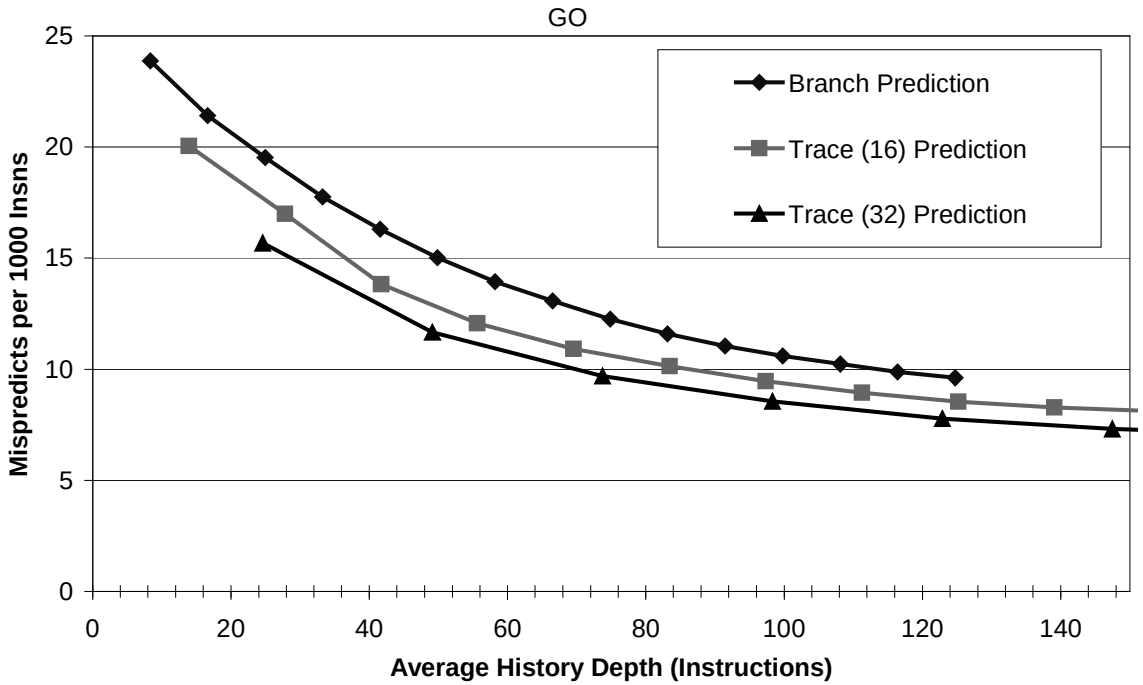


Figure 5-38 Comparisons of predictors for *go*.

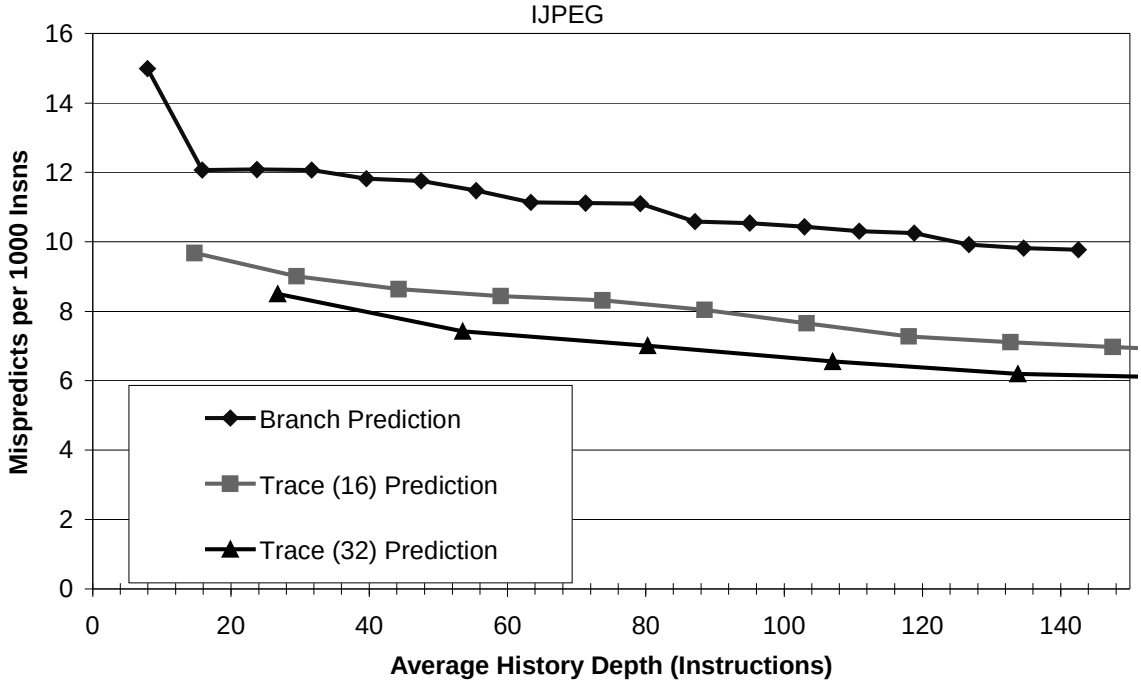


Figure 5-39 Comparisons of predictors for *ijpeg*.

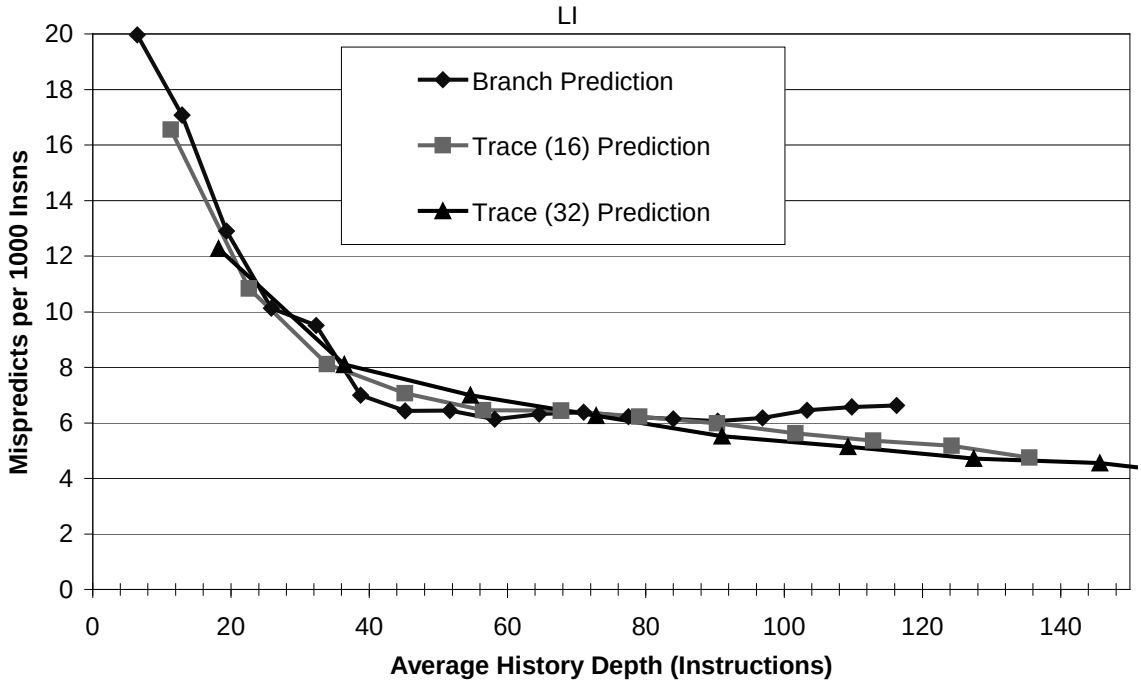


Figure 5-40 Comparisons of predictors for *li*.

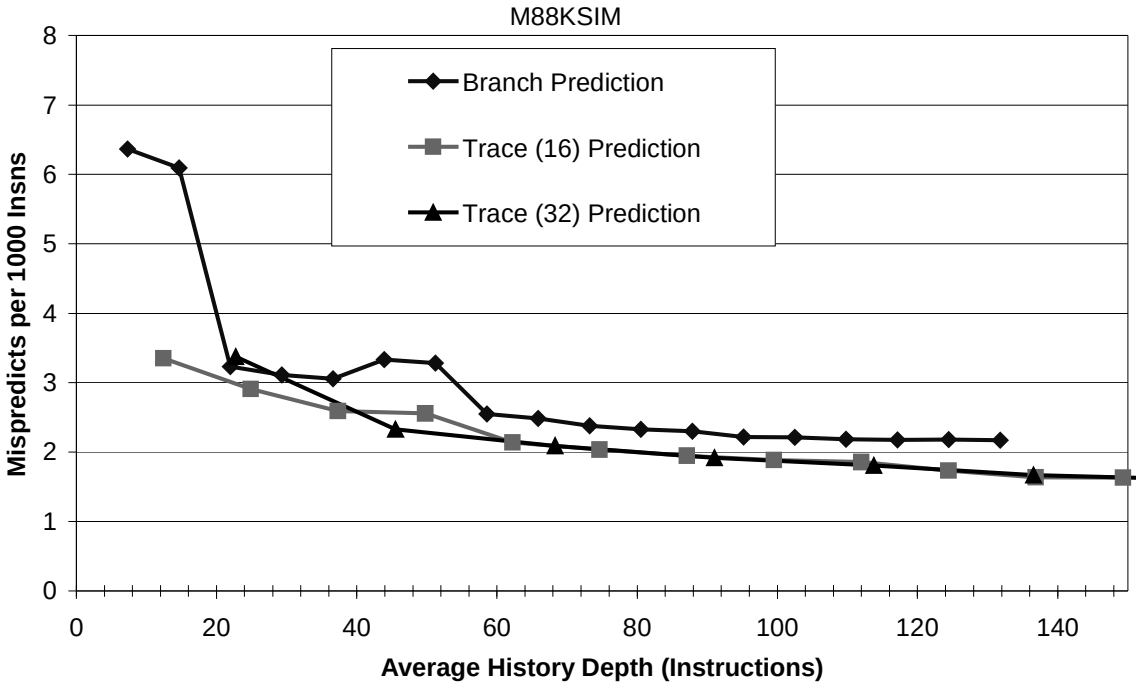


Figure 5-41 Comparisons of predictors for *m88ksim*.

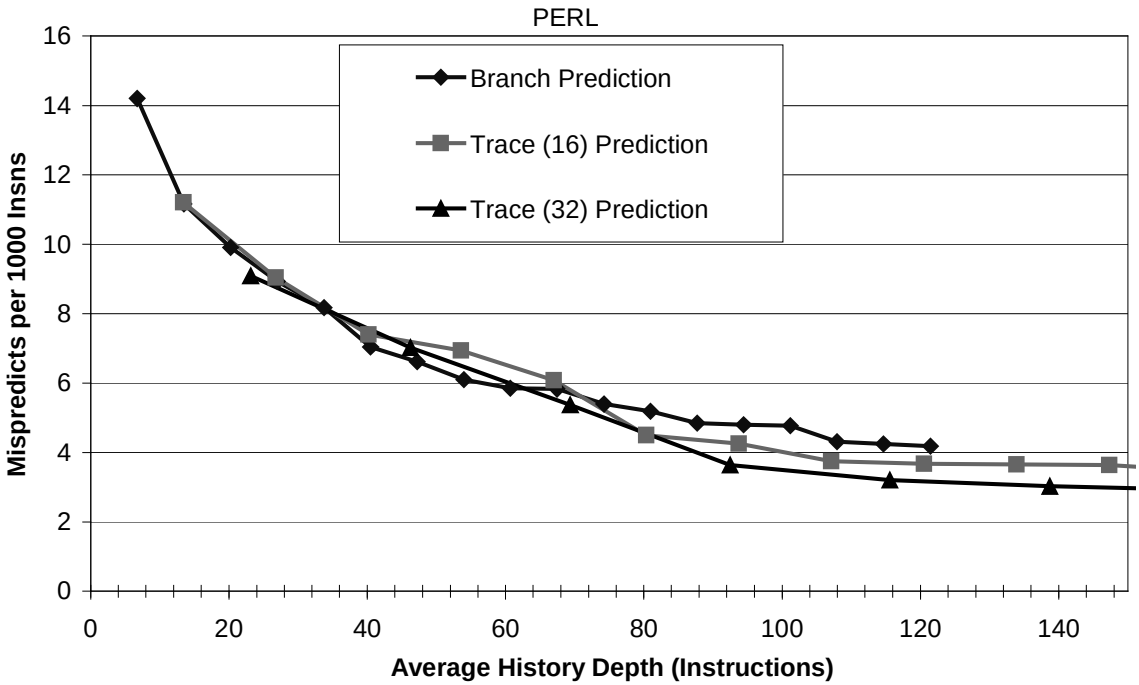


Figure 5-42 Comparisons of predictors for *perl*.

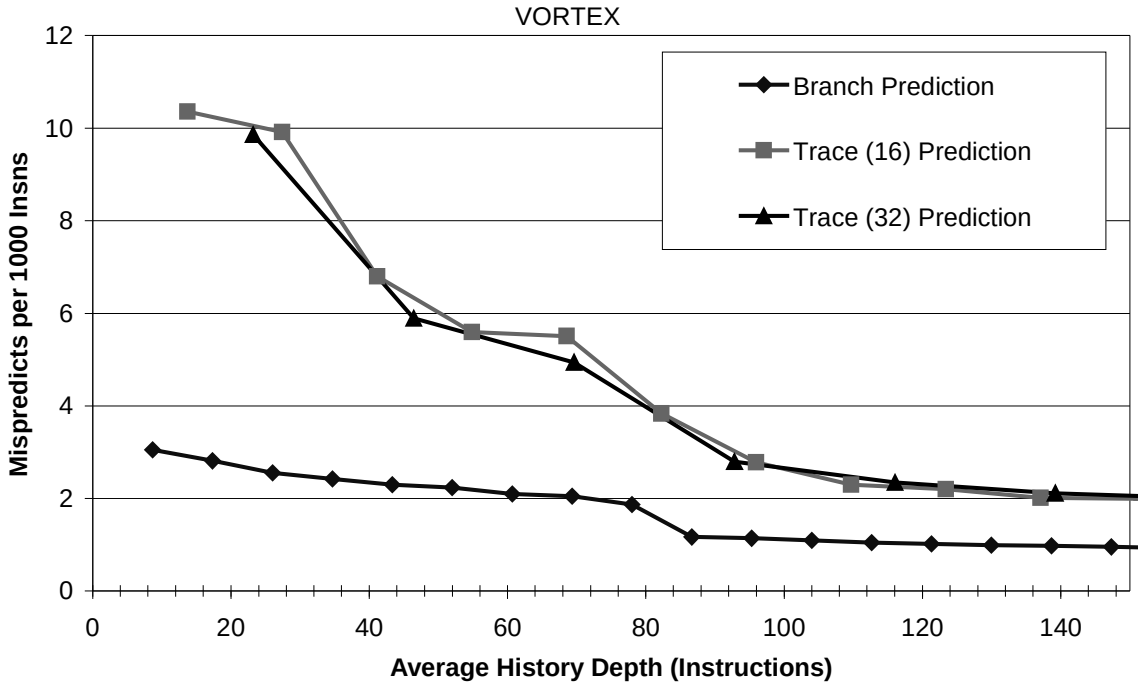


Figure 5-43 Comparisons of predictors for *vortex*.

The difference in performance between next-trace prediction and branch prediction is due to three reasons. First, trace granularity prediction is affected by trace alignment. The trace alignment provides some implicit information about what may have occurred prior to the region of code currently encoded in the history register (see Section 5.1.2). Second, the next-trace predictor implements target prediction for all control-transfer instructions as part of its normal operation, while the branch predictor is modeled with perfect branch target prediction. Finally, there is the issue of comparing statistics that are not entirely equivalent. A trace may include multiple conditional branches and if any or all of these conditional branches are mispredicted it counts as a single trace misprediction. This means that a single trace misprediction may correspond to multiple branch mispredictions in some cases.

Only in one benchmark, *vortex*, does next-trace prediction perform substantially worse than branch prediction. The performance of real branch target predictors tends to be very good, so the impact of real versus ideal target prediction is most likely minimal. This means that the reason for the lower performance of next-trace prediction is that the implicit information from trace alignment is detrimental to the predictor's performance for this benchmark.

For four of the benchmarks, *gcc*, *li*, *m88ksim* and *perl*, the performance of the next-trace predictors and branch predictor are all similar. We see that in these cases, trace granularity prediction is roughly equivalent to branch granularity prediction.

For the three other benchmarks, *compress, go* and *ijpeg*, the next-trace predictors perform significantly better than branch prediction. Also, for these benchmarks the next-trace predictor performs significantly better for length 32 traces than for length 16 traces. The differences between the predictors for these benchmarks are likely due to both the implicit information of the trace alignment and the difference in statistics, with the latter being more important. The extra information provided by trace alignment can be helpful, but there should not be a substantial difference in the amount of alignment information encoded in length 16 and length 32 traces. Trace selection includes a heuristic to limit traces to one of four alignments (for either 16 or 32 length traces) when entering a subroutine, entering a new loop iteration or exiting a loop. This trace selection will limit the amount of information that exists in the trace alignment, and make the amount of information comparable regardless of the trace length. The larger factor in the difference in performance is most likely that unpredictable branches are clustered and a single trace misprediction is occasionally

encompassing multiple branch mispredictions, with the longer traces encompassing more branches on average.

5.5 Performance for Realistic Implementations

This section quantitatively examines the performance of realistic implementations of next-trace prediction. Results are presented for all of the SPECint95 benchmarks. Corresponding results for the SEPCfp95 benchmarks are presented in Appendix A.

5.5.1 Prediction accuracy of next-trace prediction

In this section the performance of realistic implementations of next-trace predictors are explored. These implementations are based on the design proposed in Section 5.3. The realistic implementations are compared against the performance of the ideal implementation discussed in the previous section (section 5.4).

The realistic predictor is a hybrid predictor with a correlated predictor and a simple predictor. Various table sizes are considered for the correlated predictor, from 2^{14} entries to 2^{16} entries. The table of the simple predictor is 2^{15} entries (enough to nearly hold all the traces for any of the benchmarks). This table could be significantly smaller and still give comparable performance due to temporal locality. The predictor has a RHS that preserves the last trace from a subroutine when restoring the history.

The index generation functions used to implement the realistic predictor configurations are configurations that have been observed to perform well, but are not necessarily optimal. The configurations used are listed in Table 5-1.

The performance of the realistic predictor configurations is shown in Figure 5-44 through Figure 5-50 for length 16 traces and Figure 5-51 through Figure 5-59 for length 32 traces. In most cases the realistic implementations perform nearly identically to the corresponding ideal implementations. There are a few points with unexpected degradation as compared to the ideal implementation that are likely due to aliasing problems caused by the specific index generation function for that benchmark.

The two benchmarks *gcc* and *go* see consistent degradation of the realistic implementations as compared with the corresponding ideal implementations. There are also significant differences between the different sizes of correlated predictors. These two benchmarks have significantly more unique traces and more combinations of traces (as can be seen in the number of compulsory misses for the max length predictor in Figure 5-19 and Figure 5-20) than the other benchmarks. The table of the correlated predictor is therefore too small to remember all the cases and aliasing becomes a significant problem. The smaller the predictor the worse the problem. The competition for space in the table causes the performance to degrade for history lengths beyond eight.

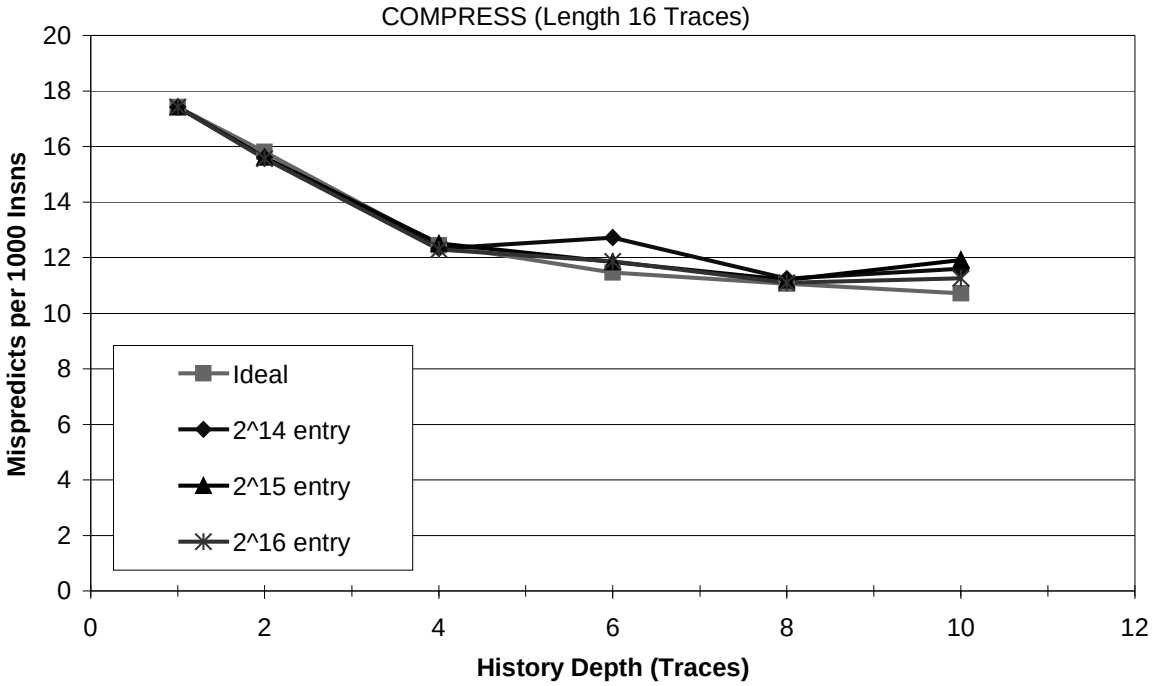


Figure 5-44 Performance of realistic predictor for *compress* with length 16 traces.

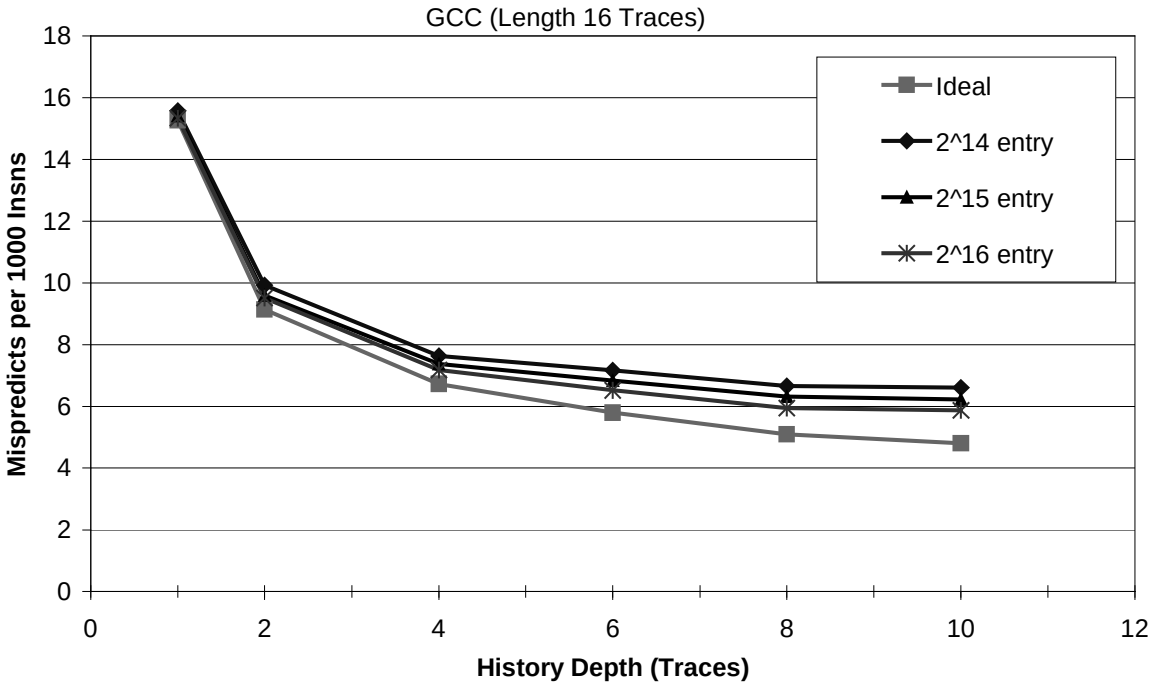


Figure 5-45 Performance of realistic predictor for *gcc* with length 16 traces.

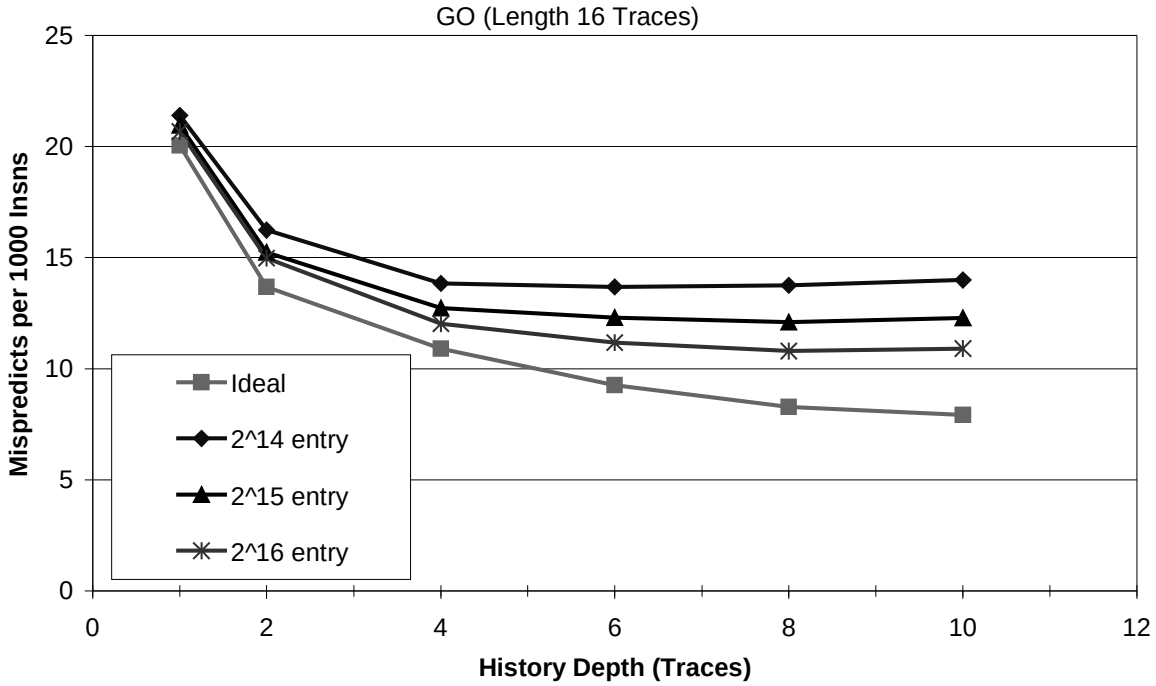


Figure 5-46 Performance of realistic predictor for *go* with length 16 traces.

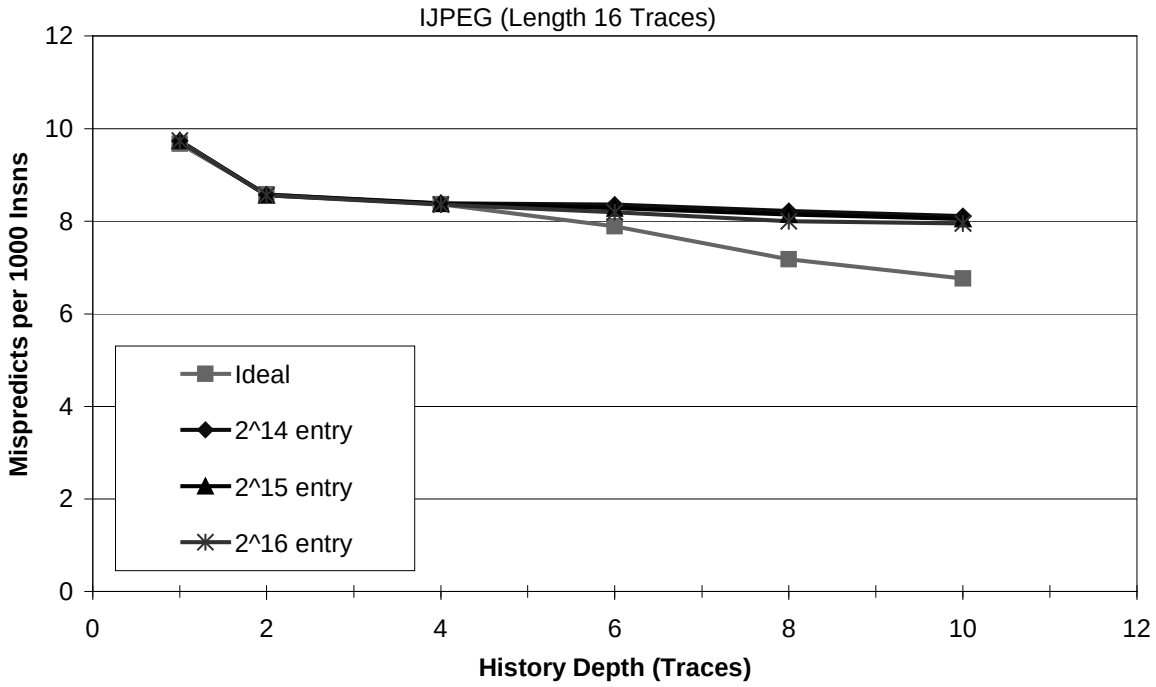


Figure 5-47 Performance of realistic predictor for *ijpeg* with length 16 traces.

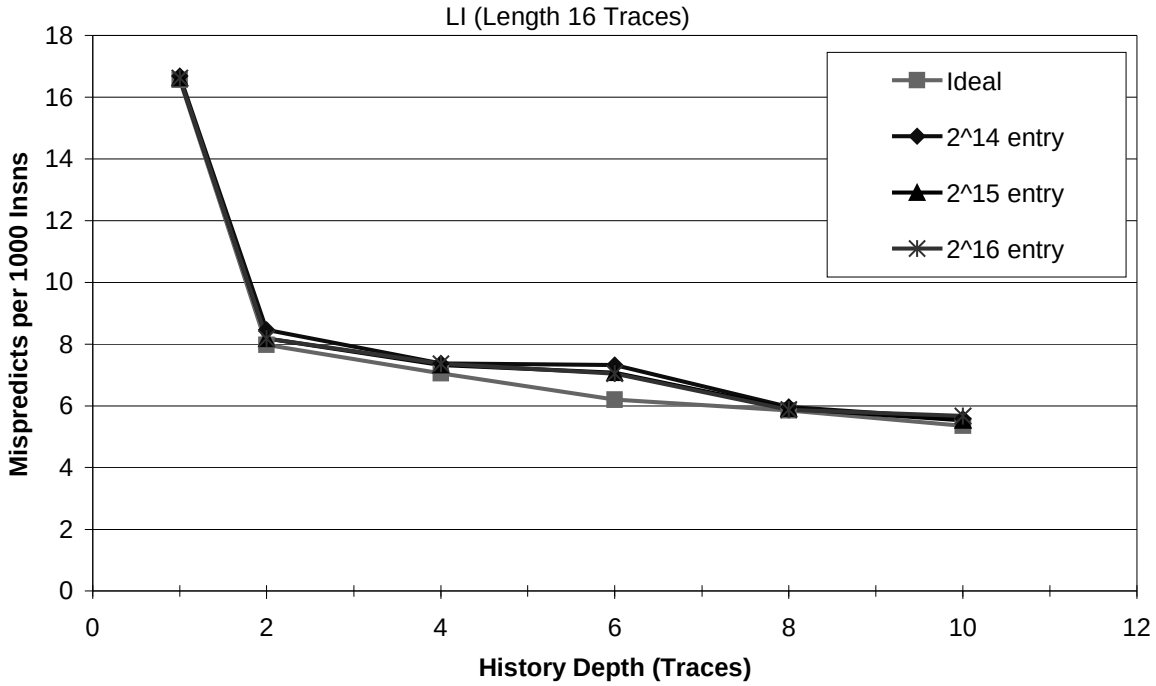


Figure 5-48 Performance of realistic predictor for *li* with length 16 traces.

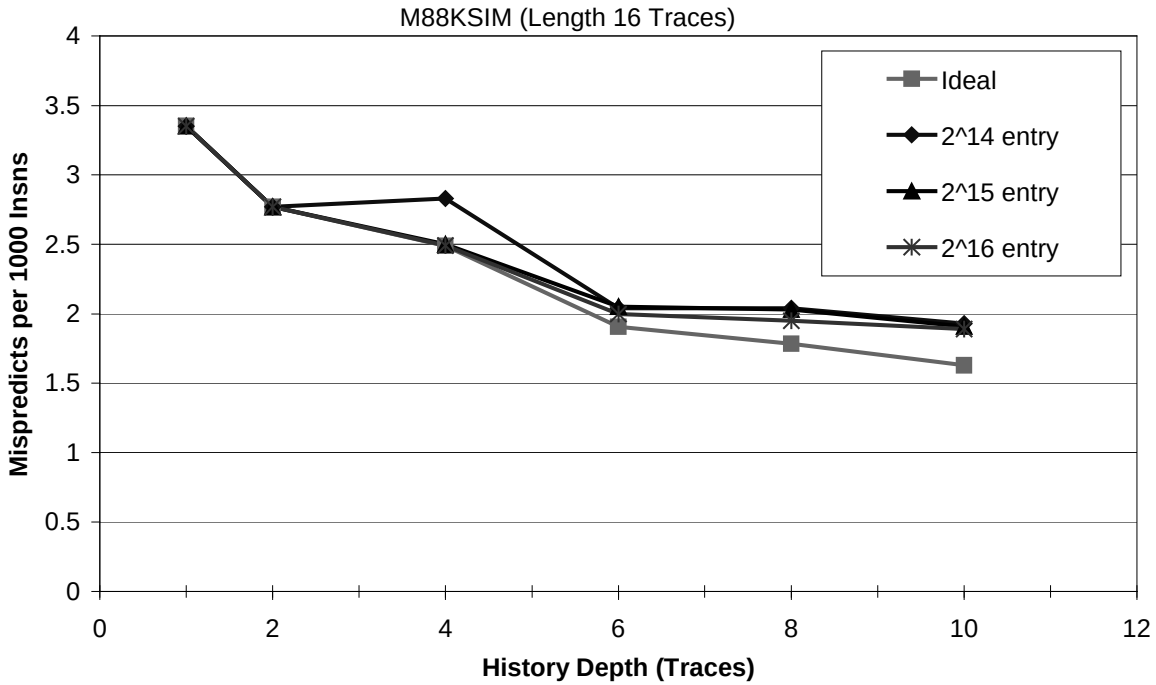


Figure 5-49 Performance of realistic predictor for *m88ksim* with length 16 traces.

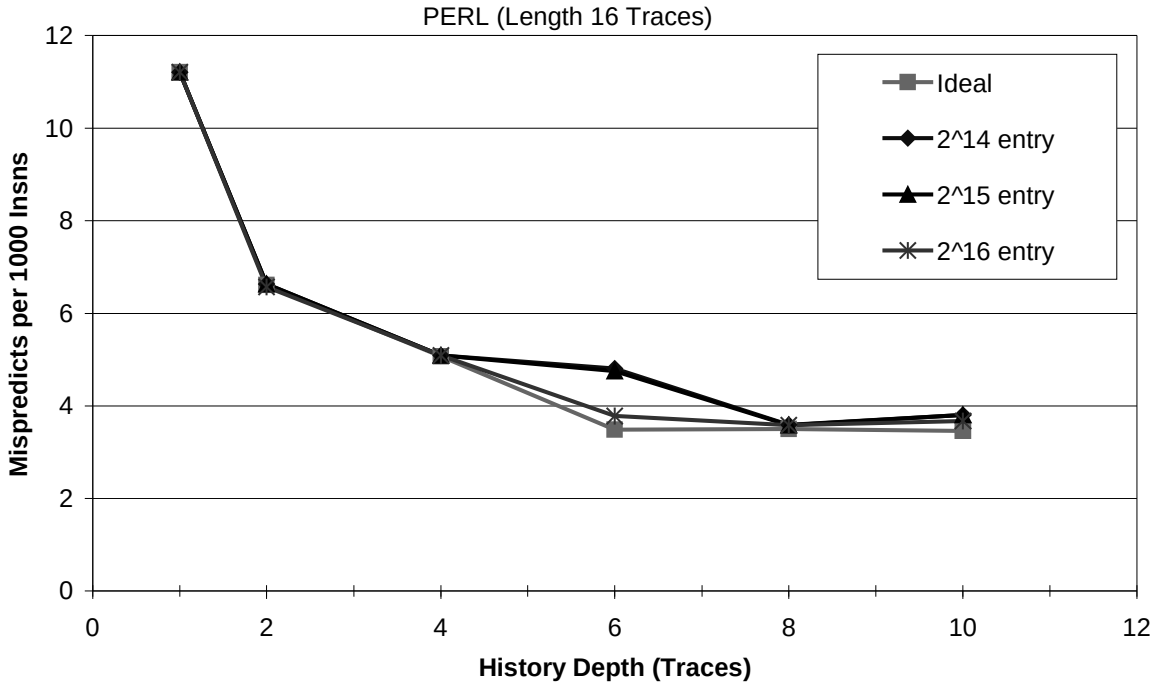


Figure 5-50 Performance of realistic predictor for *perl* with length 16 traces.

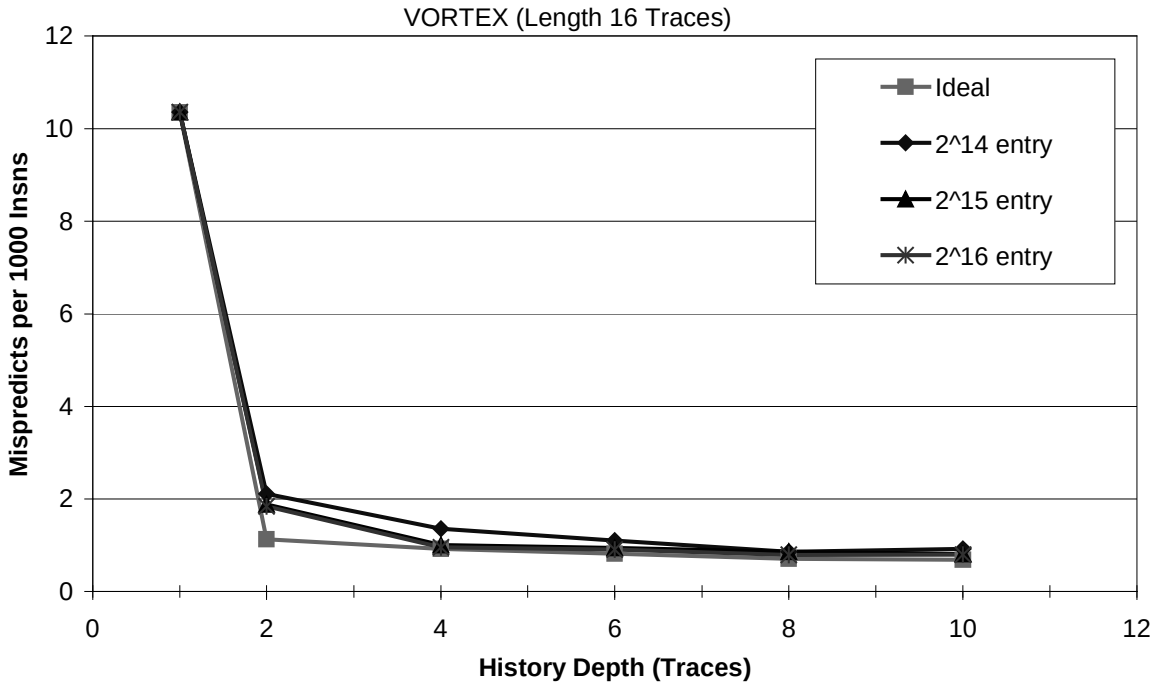


Figure 5-51 Performance of realistic predictor for *vortex* with length 16 traces.

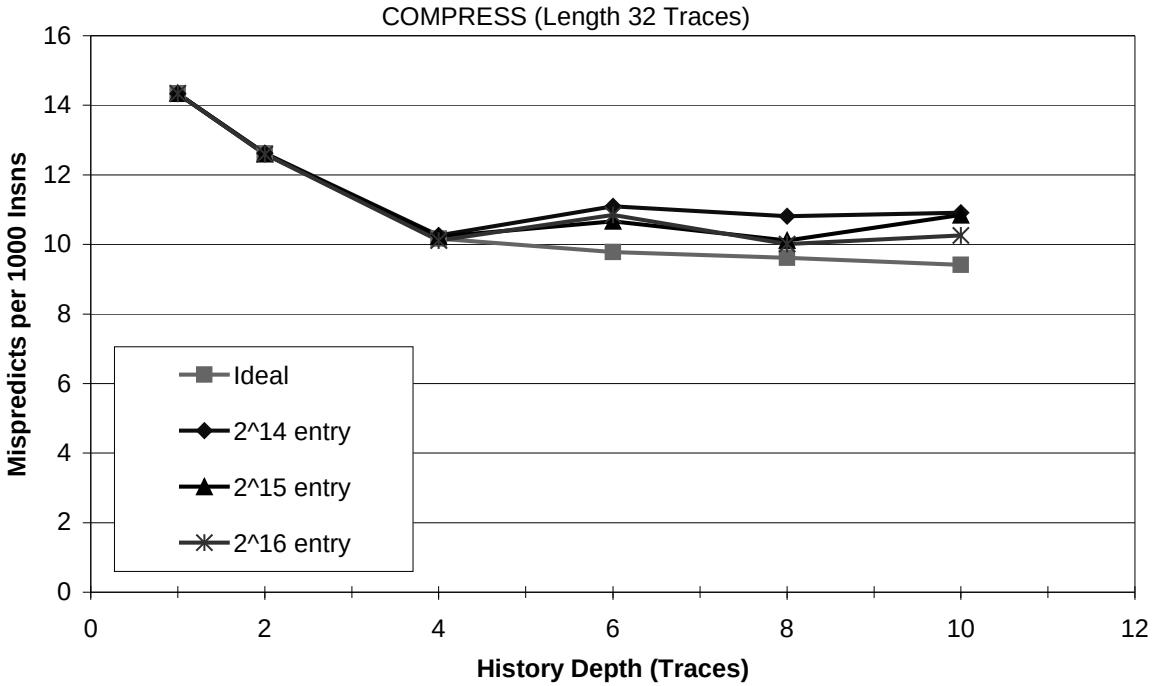


Figure 5-52 Performance of realistic predictor for *compress* with length 32 traces.

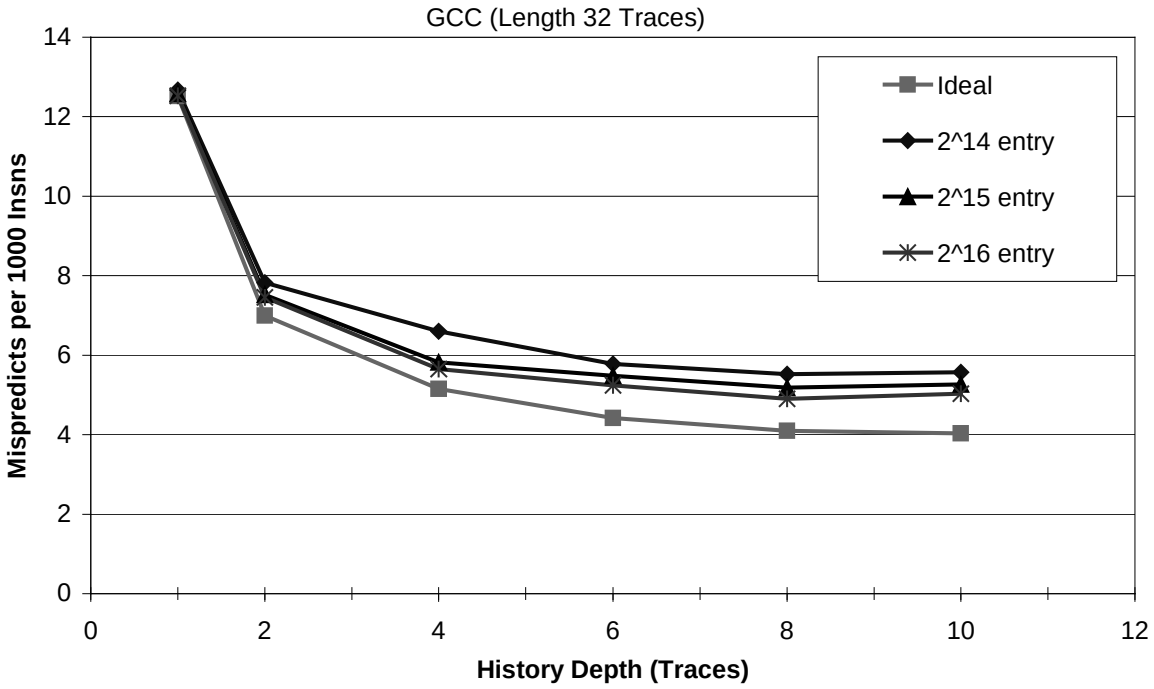


Figure 5-53 Performance of realistic predictor for *gcc* with length 32 traces.

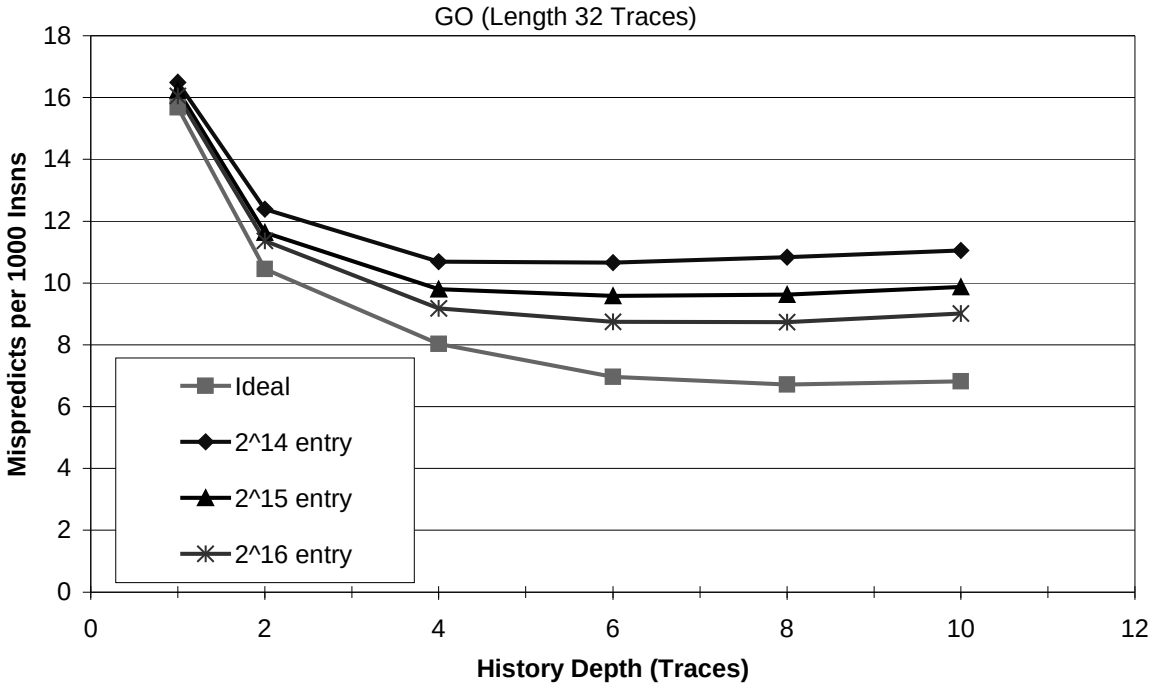


Figure 5-54 Performance of realistic predictor for *go* with length 32 traces.

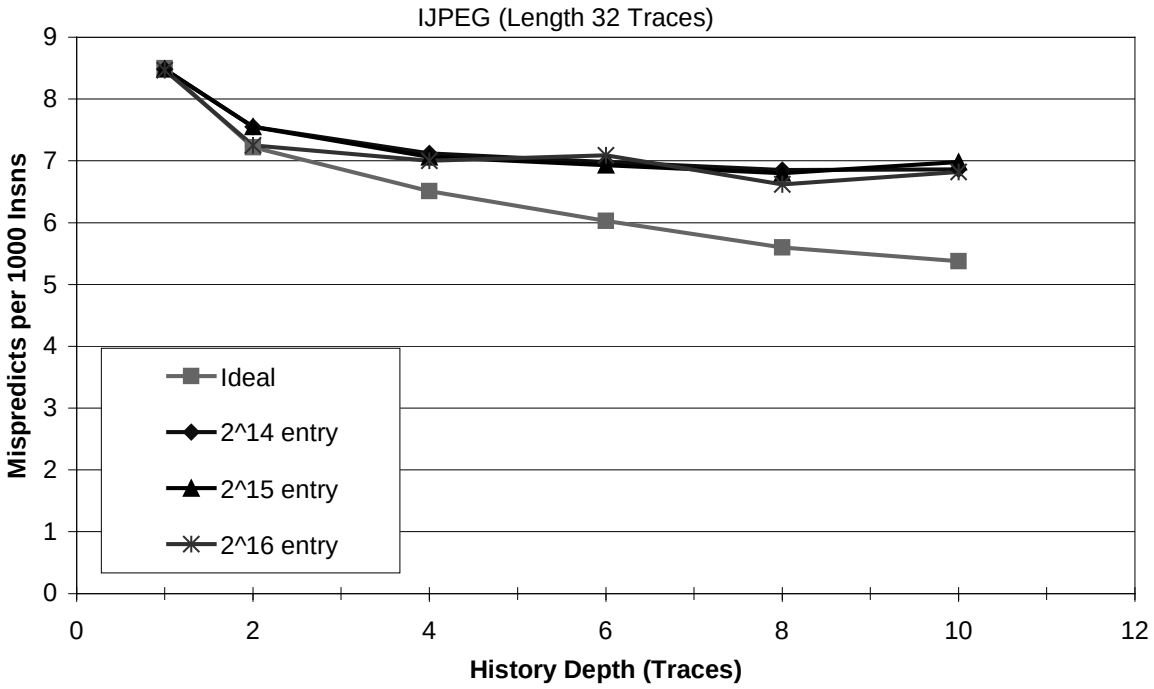


Figure 5-55 Performance of realistic predictor for *ijpeg* with length 32 traces.

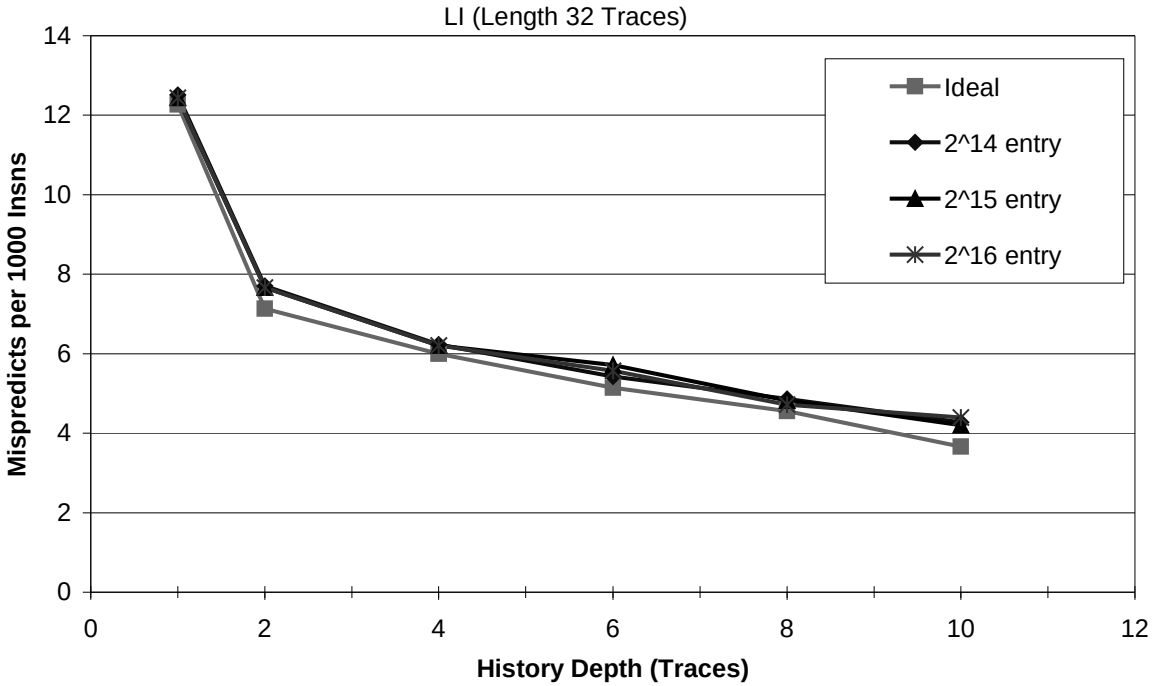


Figure 5-56 Performance of realistic predictor for *li* with length 32 traces.

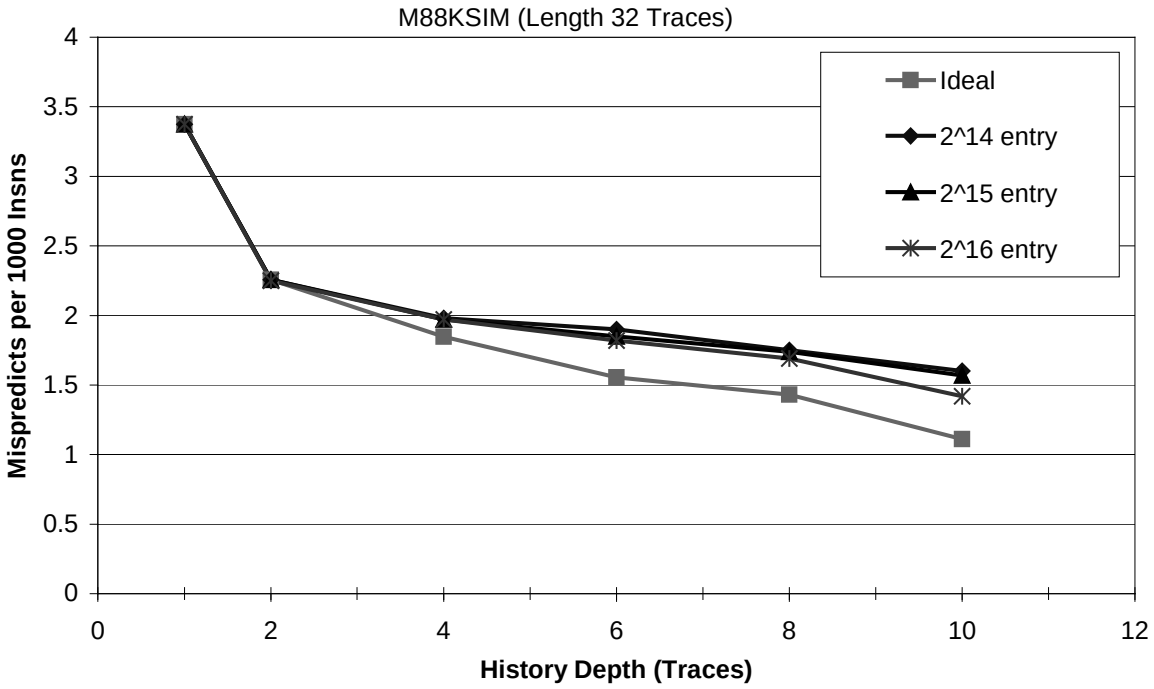


Figure 5-57 Performance of realistic predictor for *m88ksim* with length 32 traces.

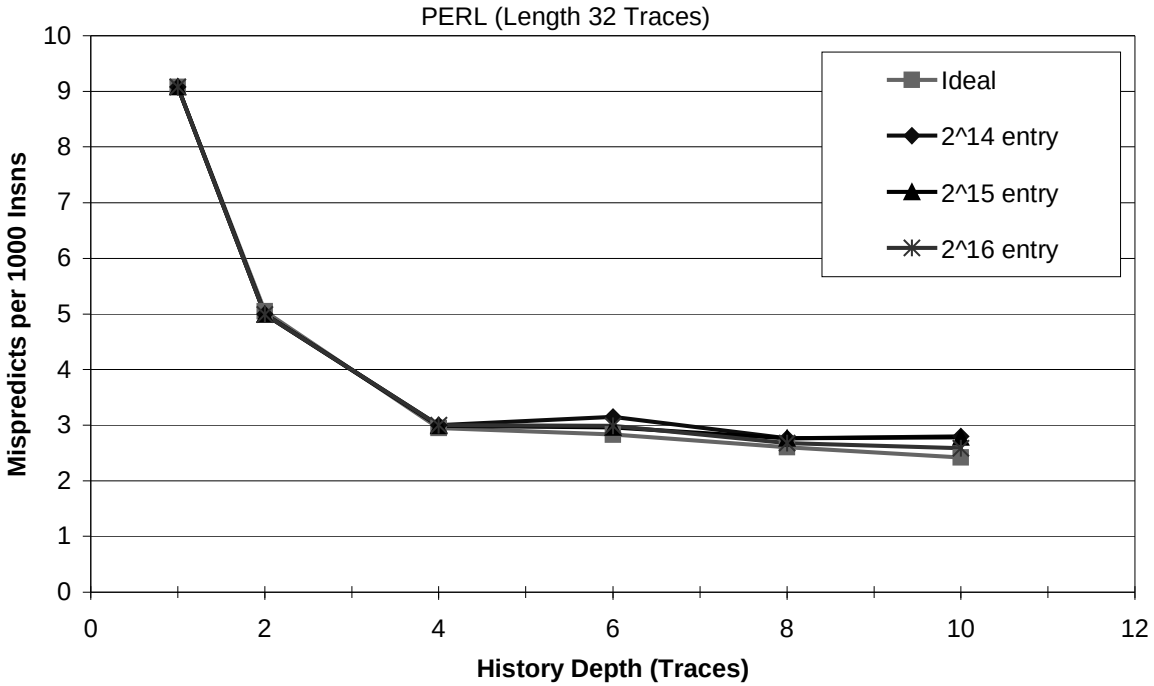


Figure 5-58 Performance of realistic predictor for *perl* with length 32 traces.

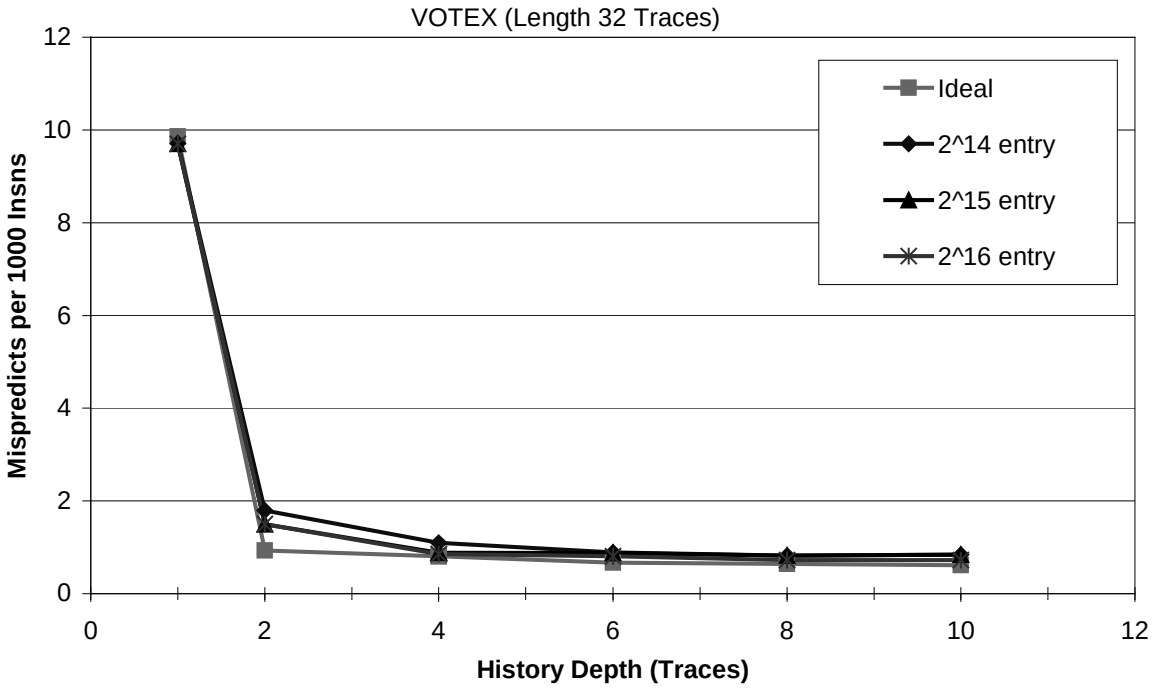


Figure 5-59 Performance of realistic predictor for *vortex* with length 32 traces.

Table 5-1 Index generation configurations used

Depth	Configuration given as: D-O-L-C (number of parts for folding)		
	14-bit Index	15-bit Index	16-bit Index
0	0-0-0-14 (1 part)	0-0-0-15 (1 part)	0-0-0-16 (1 part)
1	1-0-6-8 (1 part)	1-0-7-8 (1 part)	1-0-7-9 (1 part)
3	3-5-7-11 (2 parts)	3-5-8-12 (2 parts)	3-5-9-13 (2 parts)
5	5-3-6-10 (2 parts)	5-4-5-9 (2 parts)	5-5-5-7 (2 parts)
7	7-4-7-11 (3 parts)	7-4-9-12 (3 parts)	7-5-7-11 (3 parts)
9	9-3-7-11 (3 parts)	9-3-9-12 (3 parts)	9-4-7-9 (3 parts)

5.5.2 Impact of delayed updates

Thus far simulation results have used immediate updates. In a real processor the history register will be updated with each predicted trace, and the history will be corrected when the predictor backs up due to a misprediction. The table entry will not be updated until the last instruction of a trace has retired.

To make sure this does not make a significant impact on prediction accuracy, I ran a set of simulations with the detailed cycle simulator. The predictor being modeled has 2^{16} entries and a 7-3-6-8 DOLC configuration (the configuration used for all the timing results in this thesis). Table 4 shows the impact of delayed updates, and it is apparent that delayed updates are not significant to the performance of the predictor. For all the benchmark except *compress* the difference was less than 1%. *compress* suffers the largest degradation for real updates, but the difference is only 4% as compared to ideal updates. For compress, small loops cause the same sequences to be seen in close proximity, leading to this sensitivity. In one case, *li*, the delayed updates actually increase prediction accuracy a small amount. Delayed updates have the effect of increasing the hysteresis in the prediction table which in some cases can increase performance.

Table 5-2 Impact of real updates

Benchmark	Mispredicts with ideal updates (per 1000 instructions)	Mispredicts with real update (per 1000 instructions)	Difference
compress	11.8	12.3	4%
gcc	6.08	6.12	0.7%
go	10.9	10.9	0%
ijpeg	8.08	8.10	0.3%
li	6.50	6.49	-0.2%
m88ksim	0.07	0.07	0%
perl	3.85	4.04	5%
vortex	0.83	0.83	0%

5.6 A Cost-Reduced Predictor

The cost of the proposed predictor is primarily a function of the predictor's table sizes (most likely the correlated prediction table will be much larger than the simple prediction table). The size of each table is the number of entries multiplied by the size of an entry. The entry size of the correlated predictor is 47 bits: 36 bits to encode a trace identifier, one bit for the counter plus 10 bits for the tag. A smaller tag can be used with minimal degradation in performance, however area savings are minimal as long as the trace identifier is large.

A much less expensive predictor can be constructed, however, by observing that before the trace cache can be accessed, the trace identifier read from the prediction table must be hashed to form a trace cache index. For practical sized trace caches this index will be in the range of 10 bits. Rather than storing the full trace identifier, the hashed cache index can be stored in the predictor's tables, instead. That is, the hashing function can be moved to the input side of the prediction table to hash the trace identifier before it is placed into the table. This modification should not affect prediction accuracy significantly and reduces the size of

the trace identifier field from 36 bits to 10 bits. The full trace identifier is still stored in the trace cache as part of its entry and is read out as part of the trace cache access. The full trace identifier is used during execution to validate that the control flow implied by the trace is correct. This cost-reduced implementation is comparable in size to the multi-branch predictor proposed by Patel et al. [40].

The cost-reduced predictor should perform nearly as well as the full implementation. As long as the needed traces are in the trace cache, the behavior of the cost-reduced predictor is functionally equivalent to the full implementation. There are three problems with the cost-reduced predictor in the event that trace it references has been replaced in the trace cache. First, the cost-reduced predictor may delay the processor from recognizing the event until it is detected as a control misprediction. Second, the prediction will be incorrect if a trace is evicted from a set-associative trace cache and brought back to a different way in the set. Third, if the processor can detect that the trace is not in the trace cache, the predictor can not provide branch predictions to the slow path for generating the trace. All three of these problems can be addressed with some minor changes to the cost-reduced predictor, discussed below.

An improved implementation of the cost-reduced predictor stores the outcome of the trace's embedded branches along with the trace cache index (this only increases the size of the entry by a few bits). The outcomes can be used as a partial tag to detect most cases of the predictor referencing a trace that had been replaced in the trace cache. The outcomes can be used to perform the way select for a set-associative trace cache. The outcomes can also be used to direct the fetching of the slow path in the case of a trace cache miss.

Another enhancement of the cost-reduced predictor can help in detecting trace cache misses and performing way selection. This enhancement can be applied in addition to, or instead of, adding the outcome of embedded branches to the predictor entries. The enhancement is to add two fields to each entry of the trace cache, to hold valid starting addresses for subsequent traces (this would have a small impact on the overall size of the trace cache). Two fields would be sufficient for traces that end in any instruction other than an indirect jump (in the case of indirect jumps, any address would be considered valid). The valid starting address information can be used for both detecting most cases of the predictor referencing a trace that had been replaced in the trace cache, and for performing way selection.

With these optimizations, the cost-reduced implementation should perform almost identically to the full implementation, while still requiring considerably less area. Additional enhancements to the cost-reduced predictor could further help the performance. This is an area for future work.

5.7 Predicting an Alternate Trace

Along with predicting the next trace, an alternate trace can be predicted at the same time. This alternate trace can simplify and reduce the latency for recovering when a primary prediction is incorrect. In some implementations, this may allow the processor to find and fetch an alternate trace instead of resorting to building a trace from scratch.

Alternate trace prediction is implemented by adding another field to the correlated predictor. The new field contains the identifier of the alternate prediction. When the prediction of the correlated predictor is incorrect the alternate prediction field is updated, as follows. If the saturating counter is zero the identifier in the prediction field is moved to the alternate field and the prediction field is updated with the actual outcome. If the saturating counter is non-zero the identifier of the actual outcome is written into the alternate field.

Figure 5-60 and Figure 5-61 show the performance of the alternate trace prediction for the two representative benchmarks *compress* and *gcc*. The graphs show the misprediction rate of the primary prediction as well as the rate at which both the primary and alternate are incorrect. The results presented are for length 16 traces using a hybrid predictor with a 2^{16} entry correlated predictor based on the configurations in Table 5-1. A large percent of the mispredictions by the primary prediction are caught by the alternate prediction. For *compress*, two thirds of the mispredictions are caught by the alternate, for *gcc* it is slightly less than half. It is notable that for alternate prediction the aliasing effect quickly dominates the benefit of more history because it does not require as much history to make a prediction of the two most likely traces, so the benefit of more history is significantly smaller.

There are two reasons alternate trace prediction works well. First, there are cases where some branch is not heavily biased; there may be two traces with similar likelihood. Second, when there are two sequences of traces aliased to the same prediction entry, as one sequence displaces the other, it moves the other's likely prediction to the alternate slot. When a prediction is made for the displaced sequence of traces, and the secondary predictor is wrong, the alternate is likely to be correct.

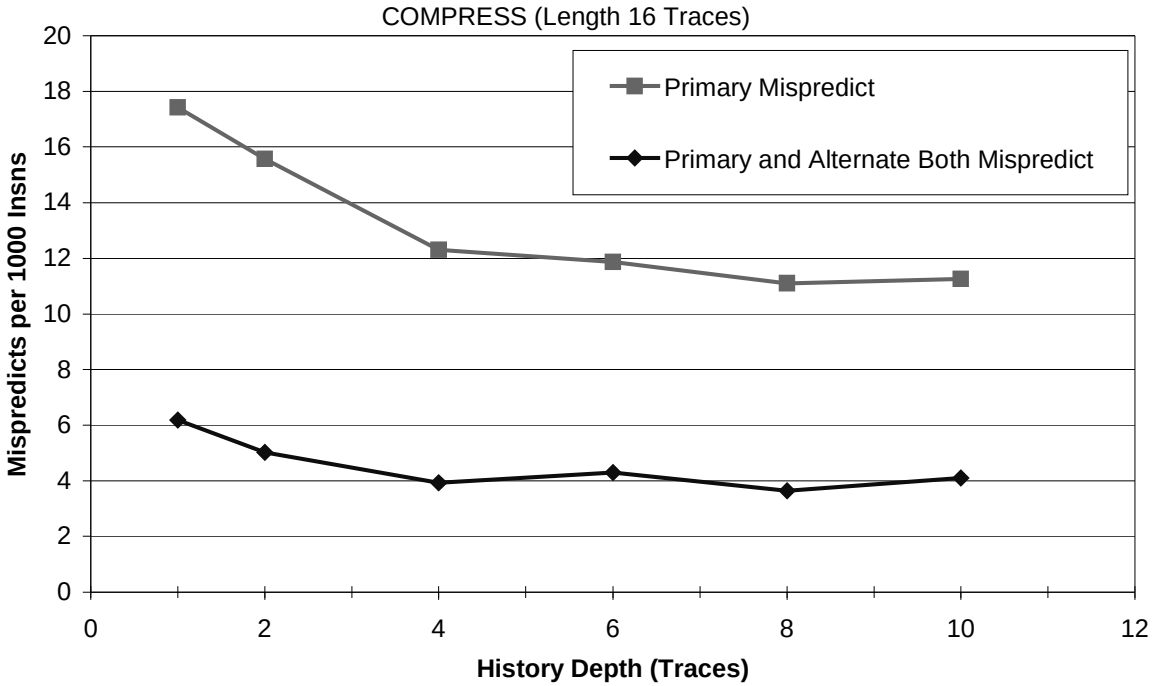


Figure 5-60 Performance of alternate trace prediction for *compress*.

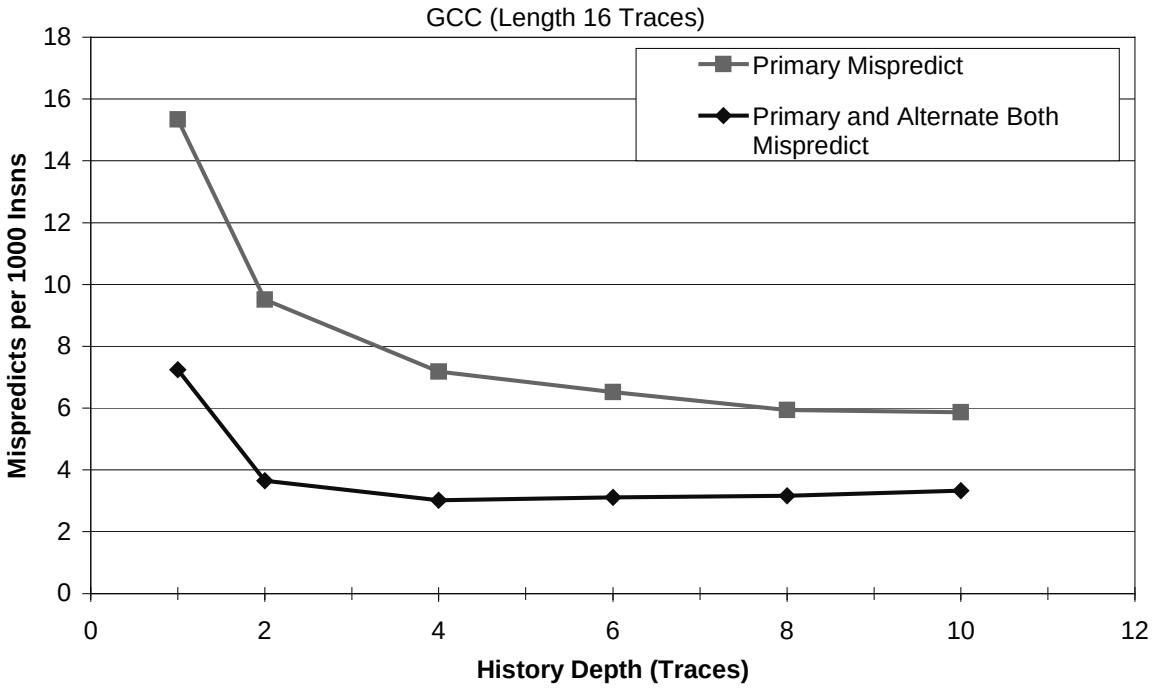


Figure 5-61 Performance of alternate trace prediction for gcc.

Chapter 6
Conclusion

The trace processor microarchitecture addresses a number of factors that are likely to become limitations for traditional superscalar architectures. The trace cache enables the processor to work in units of traces of instructions, instead of individual instructions or basic blocks of instructions. This enables high bandwidth frontends and backends to be constructed with a minimal amount of complexity and latency.

In this thesis I have proposed three mechanisms (trace pre-construction, instruction pre-processing and next-trace prediction) that work with the trace cache. Together with the trace cache, these mechanisms produce a high-performance frontend for trace processors. This frontend provides high instruction fetch bandwidth and also enables dynamic optimization of the instruction stream to expose more parallelism to the backend. The trace cache is used to decouple most of the complexity of these new mechanisms from the main processor core, in an extended pipeline organization.

This thesis introduces the concept of trace pre-construction to increase the performance of trace caches. The pre-construction mechanism sequences ahead of the processor and constructs potentially useful traces from the static program representation. Pre-construction reduced the trace cache miss rate for all of the SPECint95 benchmarks except *compress* (which sees a slight increase in miss rate). The average reduction in trace cache miss rate

was 39% (excluding *compress*). An important side effect of pre-construction is that it also prefetches lines in the instruction cache. By reducing trace cache and instruction cache misses, pre-construction produces an average speedup of 3%, and individual speedups up to 10% for applications with poor trace cache performance. The performance improvement is more substantial when pre-construction is applied in conjunction with pre-processing or other optimizations that increase the performance of the execution engine.

Pre-construction addresses the weakness of the trace cache to compulsory and capacity misses caused by the dynamic nature of traces. This thesis has shown that the more an application stresses the trace cache, the more benefits pre-construction can provide. I believe that pre-construction is necessary to enable the trace cache to scale to large real world applications that are often much larger than the SPEC benchmarks and stress the instruction fetch mechanism much more.

The implementation of pre-construction presented in this thesis represents a first attempt to perform this task. There are many ways in which pre-construction could potentially be improved (from fundamentally changing the heuristics used to stay ahead of the processor to minor changes in the way highly-biased branches are identified). I expect that future work, by myself and others, will continue to refine pre-construction. In the future, pre-construction may cease to augment the trace cache and instead replace it as the primary source for providing instructions to the processor's fetch mechanism. This would enable a small pre-construction buffer to replace the trace cache, significantly reducing the area required to implement a trace processor frontend.

This thesis has shown how the trace cache can enable a new range of dynamic optimizations. These optimizations take advantage of the intermediate program representation encoded in traces. The instructions within traces are pre-processed to optimize them for execution as a group and to optimize for implementation-specific hardware. Three specific optimizations are studied: instruction scheduling, constant propagation and instruction collapsing. Together, these optimizations produced an average speedup of 11%.

The specific pre-processing optimizations presented in this thesis are secondary in importance to the general framework that was used to implement them. This framework enables sophisticated optimizations to be incorporated with minimal impact on the main processor core. Potential future work includes developing new optimizations within this framework, as well as using this framework to implement previously proposed optimizations more efficiently. Another area of future work is explicitly getting an optimizing compiler and the pre-processing mechanism to work cooperatively to implement optimizations.

This thesis has shown that a predictor can be constructed that treats traces as basic units and explicitly predicts sequences of traces. A next-trace predictor collects histories of trace sequences and makes predictions based on these histories. The basic predictor can be enhanced to reduce performance losses due to cold starts, procedure call/return pairs, and interference in the predictor table. The predictor can achieve performance comparable to the best performance a traditional branch predictor can achieve, and significantly better than previous multiple-branch predictors have been able to achieve. The throughput of the next-trace predictor is also matched with the high fetch bandwidth of a trace cache.

The next-trace predictor is appealing, but the size of the predictor structures may be prohibitive. A cost-reduced implementation is proposed in this thesis that address this issue. Future work is needed to evaluate the performance of the cost-reduced implementation and to determine how to enable it to achieve performance comparable to the full implementation.

The future of the trace cache looks promising, but is not certain. Before final determination can be made, it is important to study the potential benefits offered by trace caches. Pre-construction and pre-processing together produce average speedup of 14%. With the introduction of pre-construction, pre-processing and other optimizations that take advantage of the trace cache, the trace cache becomes a more compelling microarchitectural feature.

Bibliography

[1] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Co., 1986.

[2] H. Akkary, M.Driscoll, "A Dynamic Multithreading Processor," in *Proceedings of the 31st International Symposium on Microarchitecture*, November 1998.

[3] S. Breach, "Design and Evaluation of a Multiscalar Processor," Ph.D. Thesis, Computer Sciences Technical Report #1393, University of Wisconsin-Madison, December 1998.

[4] W. Bucholz, ed., *Planning a Computer System*, McGraw-Hill Publishing Co., 1962.

[5] D. Burger, T. Austin and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," University of Wisconsin - Madison Technical Report #1308, July 1996. (http://www.cs.wisc.edu.~mscalar/simplescalar.html)

[6] P. P. Chang, W. Y. Chen, S. A. Mahlke, W. W. Hwu, "Comparing Static and Dynamic Code Scheduling for Multiple-Instruction-Issue Processors," in *Proceedings of the 18th International Symposium on Microarchitecture*, pp. 69-73, Nov 1991.

[7] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Water, W. W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," in *18th International Symposium on Computer Architecture*, pp. 266-275, May 1991.

[8] P.-Y. Chang, E. Hao and Y. Patt, "Target Prediction for Indirect Jumps," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 274-283, June 1997.

[9] T. Conte, K. Menezes, P. Mills and B. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates," in *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 333-343, June 1995.

[10] D. Ditzel, H. McLellan, "Branch Folding in the CRSIP Microprocessor: Reducing Branch Delay to Zero," in *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 2-9, June 1987.

[11] D. Ditzel, H. McLellan, A. Berenbaum, "The Hardware Architecture of the CRSIP Microprocessor," in *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 309-319, June 1987.

[12] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 258-263, December 1995.

[13] J.Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, C-30(7): pp. 478-490, July 1981.

[14] M. Franklin, "The Multiscalar Architecture," Ph.D. Thesis, Computer Sciences Technical Report #1196, University of Wisconsin-Madison, November 1993.

[15] M. Franklin, M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 162-171, Dec 1994.

[16] M. Franklin, G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Memory Disambiguation," *IEEE. Transactions on Computing*, pp. 552-571, February 1996.

[17]D. Friendly, S. Patel, Y. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," in *Proceedings of the 31st International Symposium on Microarchitecture*, pp. 173-181, November 1998.

[18]Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber, "Comparative Evaluation of Latency Reducing and Tolerating Techniques," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 254-263, May 1991.

[19]S. Gopal, T. N. Vijaykumar, J. E. Smith and G. S. Sohi, "Speculative Versioning Cache," in *Proceedings. of the 4th International Symposium on High-Performance Computer Architecture*, pp. 195-205, February 1998.

[20]J. Hennessy, T. Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Transactions of Programming Lang*uages *and System 5(3)*, pp. 422-448, July 1983.

[21]M. Hill, "Aspects of Cache Memory and Instruction Buffer Performance," Ph.D. Thesis, University of California at Berkeley, 1987.

[22]Q. Jacobson, S. Bennett, N. Sharma and J. E. Smith, "Control Flow Speculation in Multiscalar Processors," in *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pp. 218-229, February 1997.

[23]Q. Jacobson, E. Rotenberg, J. E. Smith, "Path-Based Next Trace Prediction," in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 14-23, December 1997.

[24]Q. Jacobson, J. E. Smith, "Instruction Pre-Processing in Trace Processors," in *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pp. 125-129, Jan 1999.

[25]D. Kaeli, P. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 34 – 41, May 1991.

[26]S. Kurlander, T. Proebsting, C. Fischer, "Efficient Instruction Scheduling for Delayed-Load Architectures," in *Proceedings of SIGPLAN '91 Conference on Programming Languages Design and Implementation*, 1991.

[27]C.-K. Luk, T. Mowry, "Cooperative Prefetching: Compiler and hardware support for effective instruction prefetching in modern processors," in *Proceedings of the 31st International Symposium on Microarchitecture*, pp. 182-193, December 1998.

[28]N. Malik, R. Eickemeyer, S. Vassiliadis, "Interlock Collapsing ALU for Increased Instruction-Level Parallelism," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 149-157, Sept 1992.

[29]D. Matzke, "Will Physical Scalability Sabotage Performance Gains," *IEEE Computer*, Volume 30, Number 9, pp. 37-39, September 1997.

[30]S. McFarling, "Procedure Merging with Instruction Caches," in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 71-79, June 1991.

[31]S. McFarling, "Combining Branch Predictors," DEC WRL TN-36, June 1993.

[32]S. Melvin, M. Shebanow and Y. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," in *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture*, pp. 60-66, Dec 1988.

[33]MIPS Technologies Inc., *R10000 Microprocessor User's Manual*, available electronically at http://www.sgi.com/processors/r10k/manual.html, 1996.

[34]R. Montoye, E. Hokenek, S. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM Journal of Research and Development*, pp. 59-70, January 1990.

[35]A. Moshovos, S. Breach, T. N. Vijaykumar and G. S. Sohi, "Dynamic Speculation and Synchronization of Data Dependencies," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 181-193, June 1997.

[36]R. Nair, "Dynamic Path-Based Branch Correlation," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 15-23, December 1995.

[37]R. Nair, M. E. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 13-25, June 1997.

[38]S. Palacharla, N. P. Jouppi, and J.E. Smith, "Complexity Effective Superscalar Processors," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 206-218, June 1997.

[39]S.-T. Pan, K. So and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," in *Proceedings of the 5th International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 76-84, October 1992.

[40]S. Patel, D. Friendly and Y. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism." University of Michigan Technical Report CSE-TR-335-97, 1997.

[41]A. Peleg, U. Weisser, "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line," U.S. Patent Number 5,381,533, Jan 1995.

[42]K. Pettis, R. Hansen, "Profile Guided Code Positioning," in *Proceedings on the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 16-27, June 1990.

[43]J. Pierce, T. Mudge, "Wrong-path Instruction Prefetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 165-175, Dec 1996.

[44]S. Przyblyski, "The Performance Impact of Block Size and Fetching Strategies," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 160-169, 1990.

[45]J. Phillips, S. Vassiliadis, "High Performance 3-1 Interlock Collapsing ALU's," *IEEE Transactions on Computers*, pp. 825-839, March 1994.

[46]D. Pnevmatikatos, M. Franklin, G. Sohi, "Control Flow Prediction for Dynamic ILP Processors," in *Proceedings of the 26th International Symposium on Microarchitecture*, pp. 153-163, December 1993.

[47]G. Reinman, T. Austin, B. Calder, "A Scalable Front-end Architecture for Fast Instruction Delivery," in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 234-245, May 1999.

[48]E. Rotenberg, S. Bennett and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24-34, December 1996.

[49]E. Rotenberg, Q. Jacobson, Y. Sazeides and J. E. Smith, "Trace Processors," in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 138-148, December 1997.

[50]Y. Sazeides, S. Vassiliadis, J. E. Smith, "The Performance Potential of Data Dependence Speculation & Collapsing," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 238-247, Dec 1996.

[51]E. Schnarr, J. Larus, "Instruction Scheduling and Executable Editing," in *Proceedings of Workshop on Compiler Support for System Software*, Feb 1996.

[52]Semiconductor Industry Association, *The National Technology Roadmap for Semiconductors*, San Jose, California, 1994.

[53]A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer* 11 (12), pp. 7-21, Dec 1978.

[54]J. E. Smith, "A Study of Branch Prediction Strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.

[55]J. E. Smith, W.-C. Hsu, "Prefetching in Supercomputer Instruction Caches," in *Proceedings of Supercomputing*, pp. 588-597, 1992.

[56]J. E. Smith, A. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," in *Proceedings of the 12th International Symposium on Computer Architecture*, pp. 36-44, June 1985.

[57]J. E. Smith, S.Vajapeyam, "Trace Processors: Moving to Fourth-Generation Microarchitectures," *IEEE Computer* Volume 30, Number 9, pp. 68-74, September 1997.

[58] G. S. Sohi, S. E. Breach, T. N. Vijaykumar, "Multiscalar Processors," in *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 414-425, June 1995.

[59]G. Tjaden, M. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers*, vol C-19, pp. 889-895, October 1970.

[60]R. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, pp. 25-33, January 1967.

[61]J. Torrellas, C. Xia, R. Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workloads," in *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pp. 360-369, January 1995.

[62]S. Vassiliadis, B. Blaner, R. Eickemeyer, "Scism: A Scalable, Compound Instruction Set Machine Architecture," *IBM Jounal of Research and Development*, pp. 59-78, Jan 1993.

[63]S. Vassiliadis, J. Phillips, B. Blaner, "Interlock Collapsing ALU's," *IEEE Transactions on Computers*, pp. 825-839, July 1993.

[64]T. N. Vijaykumar "Compiling for the Multiscalar Architecture," Ph.D. Thesis, Computer Sciences Department, University of Wisconsin – Madison, Jan 1998.

[65]S. Vijapeyam, T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 1-12, June 1997.

[66]C. Xia, J. Torrellas, "Instruction Prefetching of Systems Codes with Layout Optimized for Reduced Cache Misses," in *Proceedings of the 23rd International Symposium on Computer Architectur*, pp. 271-282, June 1996.

[67]T.-Y. Yeh, D. Marr and Y. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," in *Proceedings of the 7th International Conference on Supercomputing*, pp. 67-76, July 1993.

[68]T.-Y. Yeh and Y. Patt, "Two-Level Adaptive Branch Prediction," in *Proceedings of the 24th International Symposium on Microarchitecture*, pp. 51-61, November 1991.

[69]C. Young, E. Shekita, "An Intelligent I-Cache Prefetch Mechanism," in *Proceedings of the International Conference on Computer Design*, pp. 44-49, Oct 1993.