Trace Cache Design for Wide-Issue Superscalar Processors

by

Sanjay Jeram Patel

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1999

Doctoral Committee:
Professor Yale N. Patt, Chair
Professor Edward S. Davidson
Assistant Professor Steven Reinhardt
Professor Carl P. Simon
Joel Emer, Compaq Computer Corp

ABSTRACT

Trace Cache Design
for Wide-Issue Superscalar Processors

by
Sanjay Jeram Patel

Chair: Yale N. Patt

To maximize the performance of a wide-issue superscalar processor, the fetch mechanism must be capable of delivering at least the same instruction bandwidth as the execution mechanism is capable of consuming. Fetch mechanisms consisting of a simple instruction cache are limited by difficulty in fetching a branch and its taken target in a single cycle. Such fetch mechanisms will not suffice for processors capable of executing multiple basic blocks' worth of instructions.

The *Trace Cache* is proposed to deal with lost fetch bandwidth due to branches. The trace cache is a structure which overcomes this partial fetch problem by storing logically contiguous instructions—instructions which are adjacent in the instruction stream—in physically contiguous storage. In this manner, the trace cache is able to deliver multiple non-contiguous blocks each cycle.

This dissertation contains a description of the trace cache mechanism for a 16-wide issue processor, along with an evaluation of basic parameters of this mechanism, such as relative size and associativity. The main contributions of this dissertation are a series of trace cache enhancements which boost instruction fetch bandwidth by 34% and overall performance by 14% over an aggressive instruction cache. Also included is an analysis of two important performance limitations of the trace cache: branch resolution time and instruction duplication.

To my father.

ACKNOWLEDGEMENTS

Without the interaction of my fellow students in the HPS research group, this process would have been a much less enjoyable experience. I am indebted to them in ways I cannot fully express. In particular, I wish to thank Eric Hao for his guidance when I started out, to Marius Evers for countless hours of brainstorming and for his clarity, to Daniel Friendly for his creativity and enthusiasm, to Jared Stark, for his gifted nerdliness, and to Steven Vlaovic, who was not an HPS student, but who hung around the office enough to provide much needed distraction. There are others, and I thank them all.

Throughout the process of developing this research topic, I've had the good fortune to interact with several people who, in some way, shaped this work: Stephan Jourdan, Chien Chen, Mike Shebanow, Todd Austin, Konrad Lai, Sorin Cotofana, Stamatis Vassiliadis, and Andrea Cilio.

I also thank all those who provided (often intanglible, but incredibly important) support throughout my tenure as a graduate student. Most importantly, my mother and father and sister. Their support was invaluable. Also, Monique Thormann, Amy Weissman, Shabnam Khan, Elizabeth Rentz, Kurt Dittmar, Nanette Grace, and Chris Klein. Thank you all.

I thank our industrial partners for their financial support: Intel, IBM, and HAL. I specifically thank the Intel Foundation for providing me with a fellowship during my final year.

I am greatly indebted to my committee for providing feedback and suggestions, and making this dissertation far better than I could have made it myself.

And finally, I thank Yale Patt, my advisor and mentor. In the spirit of shorter is better, I will simply say: His influence will carry on long after I graduate.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction.

1.1 The Instruction Supply Problem

General applications possess large amounts of instruction level parallelism (ILP). Several basic studies on ILP confirm that within a contiguous window of a thousand instructions, on order of twenty instructions can be found to execute in parallel each cycle [1, 38]. This potential parallelism is the driving force behind the current design trend of building more and more execution units on a processor.

In order to make effective use of these execution units, instructions and the data needed by these instructions must be supplied at a sufficiently high rate. If ten instructions can be executed by the machine in a single cycle, then, on average, at least ten instructions must be fetched in a cycle. To do otherwise would limit performance to below the maximum potential of the hardware.

The general problem of high performance divides into three sub-problems: that of high-bandwidth instruction delivery, that of high-bandwidth data delivery, and that of high-bandwidth execution. This thesis deals with the first.

Delivering instructions at a high rate is not a straightforward task; several factors prevent the fetch engine from delivering its maximum capability. First, cache misses cause the fetch engine to stall and supply no instructions until the miss is resolved. Second, branch mispredictions cause the fetch engine to fetch instructions which will later be discarded. Third, changes in control flow inhibit the fetch engine from producing a full width of instructions. Due to the physical layout of instruction caches, taken control instructions (branches, jumps, subroutine calls and returns instruction) cause the remainder of fetched

cache lines to be discarded. With an instruction cache, it is difficult to fetch both a taken branch and its target in the same cycle.

Table 1.1 shows the average number of dynamic instructions between all branches and between taken branches (i.e., taken conditional branches, jumps, subroutine calls, returns and traps) for the SPECint95 benchmarks. The benchmarks were generated with a compiler performing standard optimizations. Evident from this table is that if partial fetches due to branches are not dealt with, then fetch bandwidth will be limited to an average upper bound of 10.52 instructions per cycle.

	cps	gcc	go	ijpeg	li	m88k	perl	vrtx	avg
insts per branch	9.15	5.34	6.21	15.59	5.54	4.24	6.06	6.57	7.34
per taken branch	13.07	8.46	9.11	19.14	8.41	5.78	9.13	11.05	10.52

Table 1.1: Instruction run lengths when terminating fetches on branches.

1.2 The Trace Cache

A straightforward technique for dealing with this limitation, called the partial fetch problem, is to construct an instruction cache where multiple fetches can be initiated at the same time. To do this, multiple addresses need to be generated, one for each block to be fetched. These addresses index into the instruction cache via multiple read ports. After the instructions are read, they must be properly aligned and merged before being supplied for execution. Such a solution adds logic complexity in a very critical execution path of the processor. Either cycle time will be affected or extra pipeline stages will be required.

This dissertation proposes the trace cache as a technique that overcomes this bandwidth hurdle without requiring excessive logic complexity in the instruction delivery path. By placing logically contiguous instructions in physically contiguous storage, the trace cache is able to supply multiple fetch blocks each cycle. A fetch block roughly corresponds to a compiler basic block: it is a dynamic sequence of instructions starting at the current fetch address and ending at the next control instruction.

Because it addresses a critical issue in high performance computer design, the trace cache has been developed concurrently and independently by several researchers [37, 41, 43].

Like the instruction cache, the trace cache is accessed using the starting address of the

next block of instructions to be fetched. Unlike the instruction cache, a line of the trace cache contains blocks as they appear in execution order, as opposed to the static order determined by the compiler. Two adjacent instructions in a trace cache line need not be at adjacent addresses in the executable image. A trace cache line stores a *segment* of the dynamic instruction stream— up to **n** instructions containing up to **m** conditional branches.

The trace cache works in concert with a branch predictor capable of sequencing through multiple control points each cycle. The predictor must be able to predict as many conditional branches each cycle as the trace cache is capable of supplying. Furthermore, the overall accuracy of this multiple branch predictor must not be significantly lower than a corresponding single branch predictor, otherwise gains in fetch bandwidth made by increasing the fetch size will be offset by more discarded fetches.

The basic objective behind this research is to increase the delivered instruction bandwidth by increasing the *effective fetch rate*. The effective fetch rate is the average width of each fetch, i.e., the average number of correct instructions issued for each fetch that returns on-path instructions. It is a direct measurement of the impact of the partial fetch problem.

1.3 The Significance of Fetch Rate

Figure 1.1 is a graph showing the relationship between effective fetch rate and overall performance, measured in instructions retired per cycle (IPC). The intent of this graph is to demonstrate a general trend. Aggressive execution hardware demands aggressive fetch.

The graph contains performance curves for three different microarchitectures as their fetch rate is artificially constrained. All three use the HPS execution paradigm in which issued instructions await operands in an instruction window. Instructions whose operands are available are scheduled for execution. The three configurations represent different levels of aggressiveness of the execution hardware, as indicated by their labels. All three mechanisms are capable of 16-wide issue. The Conservative model has a 2-cycle data cache and clustered execution units. The Aggressive model has a 1-cycle data cache, has no value bypass delays, and has ideal memory dependence detection. The Ideal model has an unlimited number of execution units and an ideal data cache. Each data point represents the harmonic mean of performance and of fetch rate for all eight SPECint95 benchmarks on a particular hardware configuration.

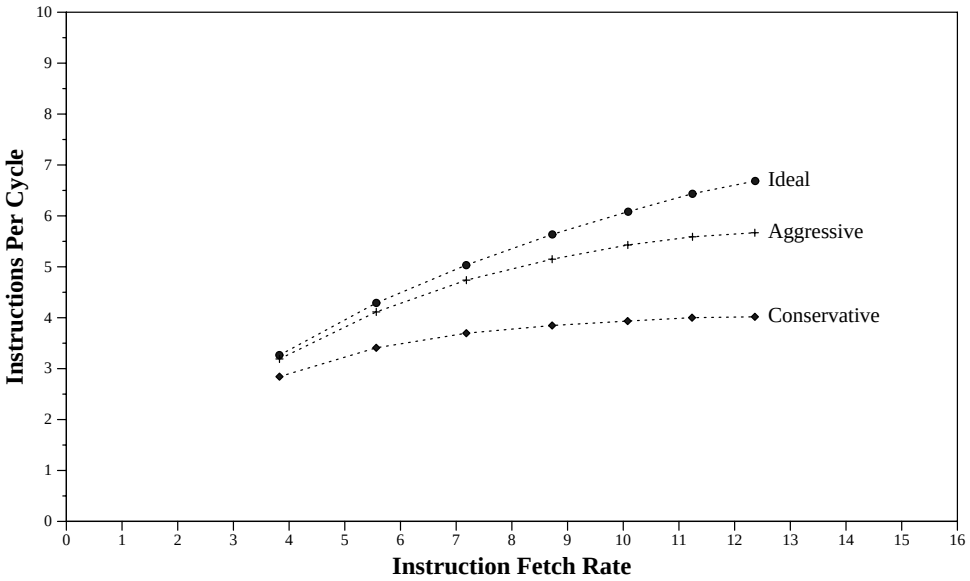

Figure 1.1: IPC as a function of fetch rate for several machine configurations.

For all configurations, performance increases as the fetch rate increases. However, the more aggressive hardware has more significant improvement. The fewer structural delays associated with instruction processing, the more pronounced the performance increase. This effect is due to the interaction of performance-inhibiting bottlenecks within the system. As one bottleneck is reduced (i.e., fetch rate), another starts to limit performance (e.g., data bandwidth).

The pursuit of high performance requires the simultaneous improvement of the processor system on many fronts. Not only must the effective fetch rate be improved, but in order for fetch rate improvements to be worthwhile, the execution hardware must be made as aggressive as possible.

Figure 1.2 demonstrates the significance of fetch rate from a slightly different viewpoint. The three curves represent the same three configurations as in the previous graph. Here, each curve shows the percent of retired instructions that could have executed earlier if they had been fetched earlier, i.e., those instruction which were fetch-limited. As the fetch rate is increased, this percentage drops. The ideal situation is one where no instructions are fetch-limited.

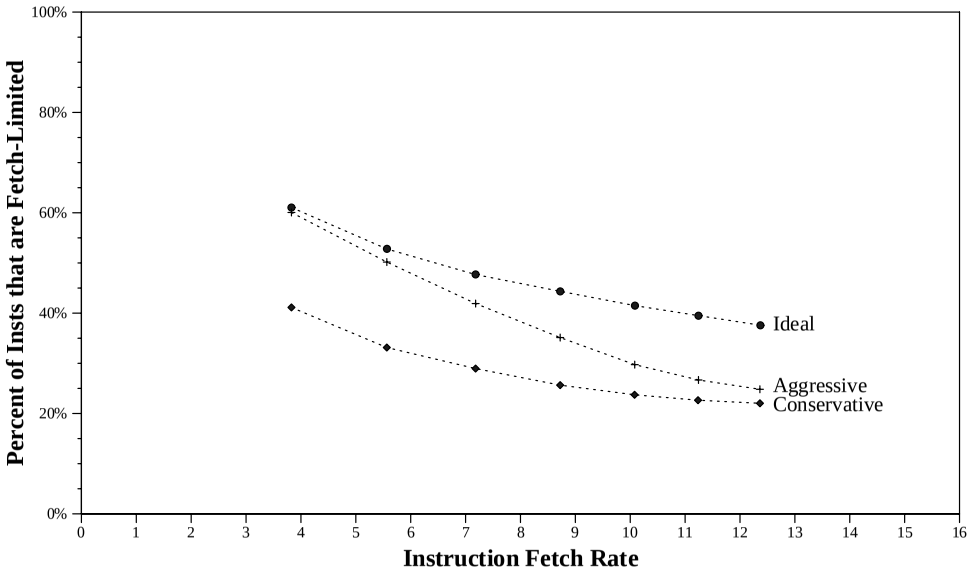

Figure 1.2: Percent of all instructions which could have executed earlier if fetched earlier. Plotted as a function of fetch rate.

1.4 Thesis Statement

In order to make effective use of a processor, the fetch mechanism must be capable of fetching as many instructions each cycle as the execution mechanism is capable of executing. Conventional fetch mechanisms are not able to deliver instruction bandwidth at a sufficient rate to support a 16-wide issue processor because of difficulty in dealing with partial fetches. The trace cache is a fetch mechanism capable of supporting a wide-issue processor.

1.5 Contributions

This dissertation makes several key contributions:

1. The trace cache fetch mechanism is described and evaluated. Overall, the trace cache boosts processor performance by 14% over that of an aggressive high-bandwidth instruction cache scheme by increasing the effective fetch rate by 34%.

2. Several improvements to the basic scheme are described. These improvements boost processor performance by 22% and effective fetch rate by 35% over a basic trace cache.

3. An analysis of fetch rate and branch resolution time reveals that as fetch rate increases, branch resolution time increases.

4. The trace cache contains on average 7 copies of an instruction. Of these, about 6 contribute to the overall delivered bandwidth of the trace cache.

5. An implementation for next-generation processor microarchitectures is described. This scheme delivers high fetch rates at high branch prediction accuracy.

1.6 Organization

This dissertation begins with a description of the instruction supply problem (this chapter, Chapter 1), shows what the partial fetch problem contributes to it, and demonstrates the significance of effective fetch rate on processor performance. The proposed method for dealing with the partial fetch problem is the trace cache and it is described in Chapter 2. Other techniques for high bandwidth instruction fetch are described in Chapter 3. The experimental framework for evaluating the trace cache is described in Chapter 4. Using this framework, the basic trace cache is evaluated in Chapter 5. An analysis presented in this chapter reveals several shortcomings of the basic scheme. In Chapter 6, several enhancements that address these shortcomings are proposed and evaluated. In Chapter 7, several configuration parameters (such as latencies and associativity) are varied to determine trace cache sensitivity on basic design factors. In Chapter 8, two effects which degrade trace cache performance are analyzed. First, as fetch rate is increased, the number of lost cycles due to branch mispredictions increases. Second, instruction duplication decreases the caching efficiency of the trace cache. In Chapter 9, a low-latency, high-bandwidth trace cache fetch mechanism applicable to next-generation superscalar processors is proposed. Chapter 10 concludes this dissertation.

CHAPTER 2

The Trace Cache Fetch Mechanism.

2.1 Overview

The basic concept behind the trace cache is that logically contiguous instructions are placed in physically contiguous storage. As blocks of instructions are fetched and executed by the processor, they are also grouped into trace segments and cached together on the same cache line. Each trace cache line contains a group of pre-processed instructions which, when fetched, can be easily formed into a *packet* for issue into the dynamic instruction window.

The trace cache fetch mechanism consists of four major components: the trace cache, the fill unit, the multiple branch predictor, and a conventional instruction cache. The trace cache is the main source of instruction supply and is filled with trace segments by the fill unit. The multiple branch predictor provides enough predictions per cycle to sequence from one trace segment to the next. The instruction cache plays an important but supporting role, handling cases where the requested instructions are not found in the trace cache. Figure 2.1 is a high-level diagram showing the four components and their relationship to each other and the other parts of the processor. Each of the four components is described in detail in this chapter. The mechanics of fetching and forming trace segments are also described.

2.2 The Trace Cache

The trace cache stores *segments* of the dynamic instruction stream, exploiting the fact that many branches are strongly biased in one direction. If block A is followed by block B

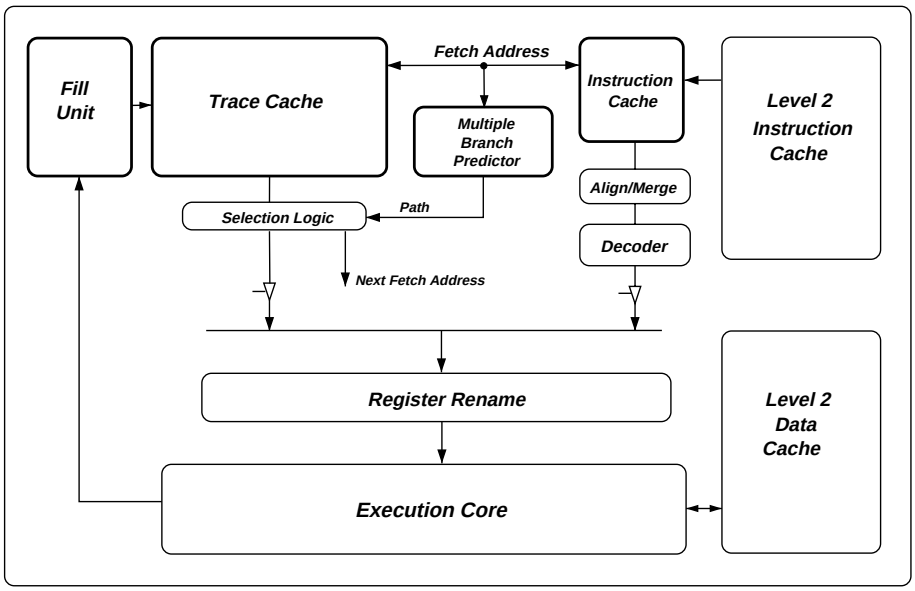

Figure 2.1: A trace cache-based processor.

which in turn is followed by block C at a particular point in the execution of a program, there is a strong likelihood that they will be executed in the same order again. After the first time they are executed in this order, they are stored in the trace cache as a single entry. Subsequent fetches of block A from the trace cache provide blocks B and C as well.

Figure 2.2 shows an example of a trace cache fetch. The address of block A is presented to the trace cache. In this example, the trace cache responds with a hit and drives out the selected segment composed of the blocks A, B, and C. The prediction structures are accessed concurrently with the trace cache. At the end of the cycle, the segment is matched with the prediction. The diagram depicts a situation where the predictor selects the sequence ABD. Since only blocks A and B match the predicted path, only A and B are supplied for execution. This is called *Partial Matching*. Its performance implications will be examined in Section 6.2.

2.2.1 Trace cache organization

In general, each trace cache segment consists of up to **n** instructions **m** of which may be conditional branches. The number of instructions is limited by the physical size of each trace cache line; the number of branches depends on the capability of the branch predictor. For example, if the branch predictor is able to predict three branches per cycle, then each trace segment can contain at most three conditional branches. The particular trace cache

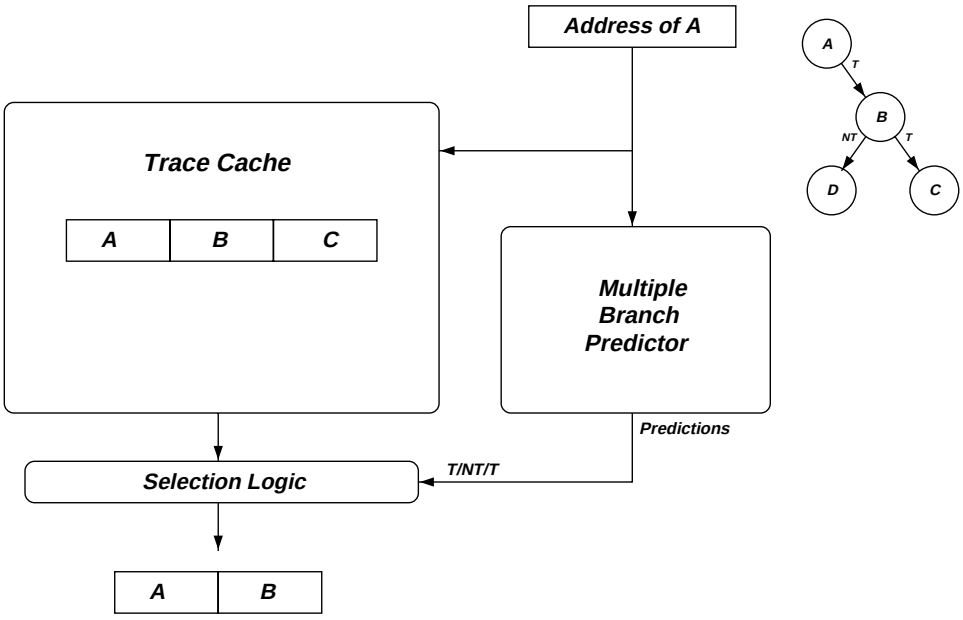

Figure 2.2: The trace cache and branch predictor are indexed with the address of block A. The inset figure shows the control flow from block A. The predictor selects the sequence ABD. The trace cache only contains ABC. AB is supplied.

implementation evaluated in this dissertation is oriented towards a 16-wide superscalar processor. The trace segments are a maximum of 16 instructions, with up to 3 conditional branches.

The organization of the trace cache is similar to that of a conventional instruction cache. The trace cache contains *lines* of instructions, which may be arranged in a set-associative manner. A fetch address indexes into the trace cache and selects a set of lines. A match is determined by comparing the tag of each line in this set with the tag portion of the fetch address. However, unlike an icache, the instructions of the matching line are not shifted, realigned, or merged (with instructions from another line) in order to form a packet after being fetched. In a trace cache, the matching line is ready for the next stage of processing (i.e., register renaming).

A trace segment is written into the trace cache only if the trace cache does not contain a longer trace segment along the same path starting at the same address. For instance, if segment ABC were resident in the cache, the new segment AB would not be added. However, if ABC were resident in the cache, the segment ABD would overwrite it. To facilitate this, the *path* information is included in the tag entry of each line. Path information encodes

the directions of all internal branches within the trace segment. If the tag indicates that a line with the same starting address already exists in the cache and the path information indicates that the line contains a larger segment along the same path, then the incoming write is not committed to the trace cache. To implement this with nominal performance impact, either a second read port is needed on the trace cache tag store or tag information is kept for each block after a trace segment is fetched. The performance impact of this trace cache write policy is evaluated in Section 7.2.

An important design option of the trace cache is whether or not to allow multiple segments starting at the same address to be resident concurrently in the cache. Allowing only one simplifies the trace segment selection logic. The implications of this design option are discussed in Section 6.3.

For the 16-wide trace cache evaluated in this dissertation, each line of the trace cache contains:

- 16 slots for instructions. Instructions are stored in decoded form and occupy approximately five bytes each for a typical ISA. Up to three branches can be stored per line. Each instruction is marked with a two-bit tag indicating to which block it belongs.

- Four target addresses. With three basic blocks per segment and the ability to fetch partial segments, there are four possible targets from a segment. The four addresses are explicitly stored allowing immediate generation of the next fetch address, even for cases where only part of the segment matches the predicted path.

- Path information. This field encodes the number and directions of branches in the segment and includes bits to identify whether a segment ends in a branch and whether that branch is a return from subroutine instruction. In the case when a segment ends with a return instruction, the return address stack (RAS) provides the next fetch address. For the reasons mentioned above, this information is stored in the tag store.

The total size of a line is around 97 bytes for a typical 32-bit ISA: 5x16 bytes of instructions, 4x4 bytes of target addresses, and 1 byte of path information.

2.2.2 Instructions are stored in decoded form

Instruction dependencies within a segment are pre-analyzed before the segment is stored in the trace cache. Each source operand identifier of each instruction includes a two-

bit tag indicating whether the source value is produced by an instruction *internal* to the segment or is produced by an *external* instruction issued in a previous cycle. If the value is produced internally, then the tag indicates which block within the segment the producing instruction belongs to. With this information, the register renaming logic can quickly determine whether the renamed tag for the source operand is supplied by the register alias table (RAT) or can be constructed without a RAT access. The destination operand identifier for each instruction is augmented with a one-bit tag indicating whether its value lives beyond its basic block, i.e., is live-out of its basic block. All values that are live-out are renamed and given a physical register. This allows the checkpoint repair mechanism, which has to create up to three checkpoints per cycle, to recover from branch mispredictions that occur in the middle of a segment without having to discard the entire segment. The concept of explicitly marking internal/external register values within a basic block was first described by Sprangle and Patt [50] and later adapted for use with the trace cache by Vajapeyam and Mitra [53].

With this additional information, each segment that is retrieved from the trace cache requires minimal renaming before being merged into the instruction window. Only instructions that have a source operand produced by an instruction outside the segment require a lookup in the RAT and only instructions that produce a value that is live-out require a physical register. Since the dependencies are explicitly indicated, complex dependency analysis of 16 instructions does not need to be performed after the segment is fetched.

Finally, instructions can be stored in an order that permits quick issue. Because the segment is pre-analyzed and the dependencies are explicitly marked, the ordering of instructions within the trace cache line carries no significance. Instructions within the cache line can be ordered to mitigate the routing required to send instructions into execution unit reservation stations. Microarchitectures may arrange the instructions within a segment to reduce the communications delays associated with incomplete bypass networks. Such a concept was evaluated by Friendly et al. [17]

2.3 The Fill Unit

The job of the fill unit is to collect instructions as they are issued by the processor and combine them into segments to be written into the trace cache. Conceptually, instructions

are presented to the fill unit as blocks in the order they were fetched. The fill unit merges the arriving blocks with awaiting blocks latched in a previous cycle. The merge process involves maintaining dependency information and reordering instructions. The process continues until the segment becomes *finalized*, at which point it is enqueued to be written into the trace cache.

Dependency information is maintained by recalculating the two-bit source operand dependency tag on each arriving instruction. This tag is changed to reflect whether an awaiting instruction generates the value needed by the incoming instruction.

A segment is finalized when

1. it contains 16 instructions, or
2. it contains 3 conditional branches, or
3. it contains a single indirect jump, return, or trap instruction, or
4. merging the incoming block would result in a segment larger than 16 instructions.

Rule 1 is implied by the size of the trace cache line and rule 2 by the capability of the predictor. Because their targets vary, return instructions and indirect jumps cause finalization (rule 3). Unconditional branches and subroutine calls are not a factor in trace segment finalization.

The default fill unit strategy treats fetch blocks as atomic entities (rule 4). A fetch block is not split across two segments unless the block is larger than 16 instructions. The rationale for this strategy and the implications of relaxing it are examined in Section 6.7.

Three outcomes are possible with the arrival of each new block of instructions: (1) The arriving block is merged with the unfinalized segment and the new, larger segment is not finalized. (2) The entire arriving block cannot be merged with the awaiting segment. The awaiting segment is finalized and the arriving block now occupies the fill unit. (3) The arriving block is completely merged with the awaiting segment and the new, larger segment is finalized.

The fill unit can collect blocks as they are issued into the instruction window or as they are retired. If blocks are collected at retire time, segments due to speculative execution are not added to the cache, potentially reducing misses. On the other hand, creating segments on the wrong path may generate segments that may be useful later. The additional lag time

in generating segments after retirement might also have negative effect on performance: the first few iterations of a tight loop will miss until the first iteration is retired and written into the trace cache. The effects of this design issue are examined in Section 7.6.1.

2.4 The Branch Predictor

The branch predictor is a critical component of a high bandwidth fetch mechanism. To maintain a high instruction bandwidth, the branch predictor needs to make multiple accurate branch predictions per cycle. Predicting only a single branch would introduce a one branch per cycle bottleneck. In the case of the trace cache mechanism presented here, three predictions per cycle are required.

Two-level adaptive branch prediction has been demonstrated to achieve high prediction accuracy over a wide set of applications [55]. In a two-level scheme, the first level of history records the outcomes of the most recently executed branches. The second level of history, stored in the pattern history table (PHT), records the most likely outcome when a particular pattern in the first level history is encountered. In typical schemes, the pattern history table consists of saturating 2-bit counters.

The trace cache branch predictor uses the two-level scheme called *gshare* (see McFarling [29]). The global branch history is XORed with the current fetch address, forming an index into the PHT. Gshare has been shown to be effective in reducing negative interference in commonly accessed PHT entries. McFarling has demonstrated the increased accuracy of gshare over other global history based prediction schemes.

To make three predictions per cycle, we expand each PHT entry from a single two-bit counter into three two-bit counters, each two-bit counter providing a prediction for a single branch. Variations of this scheme are presented in Section 7.7. A diagram of this predictor is presented in Figure 2.3.

2.5 The Instruction Cache

A conventional instruction cache supports the trace cache by supplying instructions when the trace cache does not contain the requested segment. If hitting in the trace cache is the frequently occurring case, then the supporting fetch mechanism need not be designed for higher bandwidth. In Section 7.5, several instruction cache configurations of varying

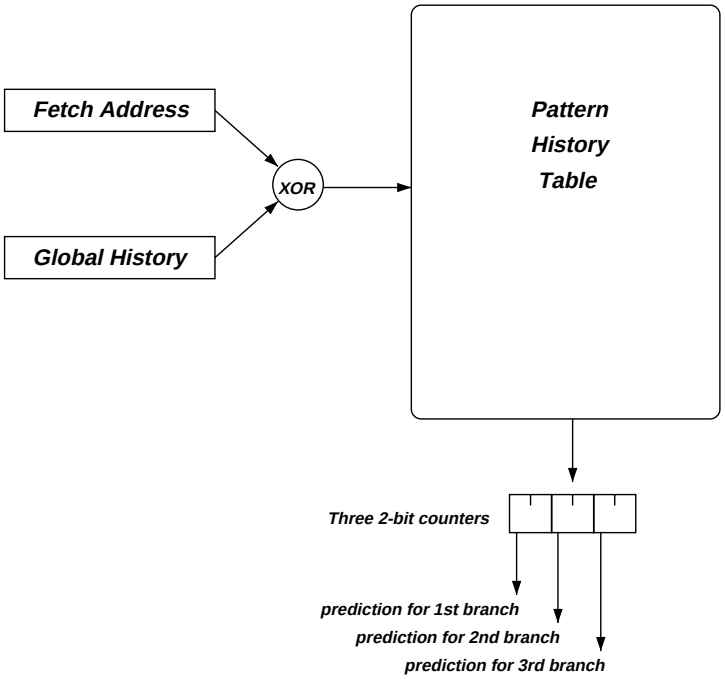

Figure 2.3: The multiple branch predictor supplies 3 predictions per cycle using a variant of the gshare scheme developed by McFarling.

aggressiveness are evaluated to test this hypothesis.

Regardless of the aggressiveness of the instruction cache, the instructions fetched from it must be properly aligned and merged in order to form an issue packet. The packet must then be decoded and renamed before it is added to the instruction window.

In the case of the trace cache fetch mechanism, the instruction cache supplies up to one fetch block per cycle. The instruction cache has two read ports to allow adjacent cache lines to be retrieved each cycle. By fetching two cache lines and realigning instructions, the icache mechanism overcomes partial fetches due to cache line boundaries.

2.6 The Fetch Cycle

At the beginning of the cycle the fetch address, determined in the previous cycle, is presented to both the trace cache and the instruction cache. The fetch address is also used by the branch predictor, along with global branch history, to index into the predictor's pattern history table. Some time into the cycle, the trace cache will respond with either a hit and the matching cache line, or a miss. The branch predictor will respond with three predictions. The instruction cache will produce the matching cache line(s), along with a

hit/miss signal.

In the case of a trace cache tag hit, the selection logic determines which portion of the matching line to issue. It does so based on a comparison of the path information of the matching line and the branch predictions produced that cycle. Selection can be conservative: if the entire path of the fetched trace segment does not match the predicted path, then nothing is issued (i.e., it is equivalent to a trace cache miss). The technique can be more aggressive: only the matching portion is issued. This selection technique is evaluated in Section 6.2.

The branch predictions also select which of the four possible fetch addresses to use for the following cycle. If the fetched segment ends in a return, the next fetch address is supplied by the return address stack. The branch predictor's history register is updated, shifting in the appropriate predictions for the speculatively fetched branches. Several alternatives for managing the timing problem associated with selecting the next fetch address are discussed in Section 6.2.1.

Selecting the proper next fetch address and updating the global history register are the critical operations which need to be completed in the fetch cycle. If only a portion of the provided segment matches, unwanted instructions can be invalidated in the next stage of processing.

In the case of a trace cache miss and an instruction cache hit, up to a single fetch block is supplied to the decoder at the end of the cycle. If both the trace cache and instruction cache miss, then a request for the missing instruction cache line is made to the second level cache. The fetch mechanism stalls until the missing line arrives.

CHAPTER 3

Related Work

3.1 Overview

The trace cache is a hardware-based scheme for increasing the instruction fetch rate. In this chapter, the history of the trace cache is described, along with more recent research in expanding the role of the trace cache.

However, the trace cache is not the only means of high-bandwidth fetch. The problem of high fetch rate has been attacked on two fronts. First, several hardware-based schemes have been proposed, most of which address the problem by using a multi-ported or interleaved instruction cache to which multiple independent requests are made. Second, several compiler-based schemes have also been proposed, most of which use profiling to determine the likely direction of branches and then build large sequential blocks by laying out the likely path as the fall-through path. In this chapter, these other hardware schemes and compiler schemes are described.

3.2 The History of the Trace Cache

The roots of the trace cache can be traced back to the original introduction of the fill unit by Melvin at al. in 1988 [32]. Although not explicitly intended to create segments of the dynamic instruction stream, the proposed fill unit saves the work done by the decoder of an HPS [40] implementation of the VAX architecture. In this manner the fill unit (and associated caching structure) overcomes the decoder bottleneck associated with decoding a complex instruction set such as the VAX architecture. In a subsequent paper published in

1989, Melvin and Patt [31] first propose using the fill unit mechanism to store two or more basic blocks worth of instructions in an entry in the associated cache. Since Melvin and Patt were investigating 4-wide machines at the time, for which high-bandwidth fetch is not a necessary component, the concept was not pursued further.

The basic concepts described by Melvin et al. were explored further by Smotherman and Franklin. In their first paper [15], they applied the original fill unit concept to dynamically create VLIW instructions out of RISC-type operations. They reworked the fill unit finalization strategy by restricting the type of instruction dependencies allowed in a fill unit line and by filling both paths beyond a conditional branch. Their second paper [48] describes how a fill unit can help overcome the decoder bottleneck of a Pentium Pro processor.

The first documented evaluation of the trace cache appeared as a US patent filed in 1994 by 'Intel Corporation, with Peleg and Weiser as inventors [41]. They called the caching structure the Dynamic Flow Instruction Cache, and each line of it contains two blocks from the dynamic instruction stream, along with a prediction for each branch. They propose using each dynamic block as a starting point for each trace segment created. This is different from the current trace cache schemes where a new segment begins where the previous segment ended.

Johnson developed the Expanded Parallel Instruction Cache (EPIC) in 1994 [26]. The Expansion Cache in his scheme is very similar to the trace cache concept, but targeted for a statically scheduled microarchitecture. Each line of the Expansion Cache contains instructions in dynamic order, across a single branch. These instructions are stored in a manner to make processing them easier once they have been fetched: they are decoded, pre-analyzed, and pre-routed to execution units, similar to the trace cache proposed in this dissertation. The objective of the Expansion Cache is that of moving processing complexity to the cache fill pipeline. The resulting machine is a statically-scheduled machine which can adapt to run-time variations.

Rotenberg et al. [43] evaluated the trace cache concept in 1996 as a mechanism to deliver high instruction bandwidth to a wide-issue machine. They demonstrated the trace cache's lower latency (because of lower complexity) than other hardware based schemes for high-bandwidth instruction fetch. They demonstrated that the trace cache was more effective than the Branch Address Cache and the Collapsing Buffer (both described in section 3.4.1) at instruction supply. Their mechanism consisted of a large instruction cache and a small,

supporting trace cache.

Patel et al. concurrently developed the trace cache concept in 1996 and published their findings in [37]. In this initial work, they demonstrate some techniques which significantly increase the instruction bandwidth delivered by the trace cache. Furthermore, they describe an effective multiple branch predictor design. They show that a large trace cache configuration can significantly outperform an instruction cache, even if the instruction cache is equipped with state-of-the-art branch prediction.

3.3 Trace Cache Extensions

Since its introduction into mainstream microarchitecture research in 1996, the trace cache has been extended in both application and performance by several researchers.

Patel et al. [16, 35–37] have significantly increased the delivered fetch rate of the basic trace cache proposing enhancements which systematically address limitations to its performance. Partial Matching and Inactive Issue were first explored in 1997 [16], and Branch Promotion and Trace Packing in 1998 [35].

Rotenberg et al. [44] have pursued an equally important, and complementary line of trace cache research. They have explored the superscalar processor implications of using a trace cache. They demonstrate how hardware complexity can be greatly reduced with the trace cache without significant compromise in performance. Their microarchitectural model is *trace-centric*, and the basic unit of processing is a trace segment. The basic ideas behind their approach are (1) fast value communication is most important within a trace segment, so one trace segment is allocated within a cluster of execution units where communication between instructions can be done without extra latency, and (2) speculative sequencing is best performed at the trace level i.e., individual branches need not be predicted, only next traces. To go along with their trace processor approach, a next trace predictor was developed by Jacobson et al. [24]

The concept of dynamic trace optimizations was first explored by Friendly et al [17]. With dynamic trace optimizations, the fill unit optimizes a trace segment by applying aggressive compiler-style optimizations (such as constant reassociation) and microarchitectural targeting (such as smarter scheduling of instructions to avoid communications delays). Similar concepts were concurrently investigated by Jacobson and Smith [25].

Along the lines originally developed by Johnson and the Expansion Cache, Nair and Hopkins explore the idea of using a mechanism to build and cache statically scheduled instruction groups [34]. The concept, called the Dynamic Instruction Formatting (DIF) cache, contains dynamically created VLIW-like instructions which can then be fed to a high-speed statically-scheduled engine without instruction set support.

Several researchers have also proposed auxiliary extensions to the trace cache. Vajapeyam and Mitra [53] adopt a scheme for explicitly naming register dependencies to the trace cache and suggest a scheme for dynamically creating vector-like instructions out of loops which are completely encoded within a trace segment.

3.4 Other Techniques for High-Bandwidth Instruction Fetch

3.4.1 Hardware-based techniques

One of the simplest forms of hardware-based high bandwidth fetch mechanisms is a sequential block scheme where fetching continues past branches which are predicted to be not taken. With this scheme, fetch bandwidth is limited by taken branches and predictor bandwidth. When this scheme is coupled with an optimizing compiler which performs profile-guided code layout to reduce taken branches, the delivered fetch bandwidth is considerably high, at a low hardware cost. Since this scheme offers high performance at low-complexity, it is evaluated as a comparison to the trace cache in this dissertation.

One of the earliest hardware-based multiple block fetch schemes was proposed by Yeh et al. [54]. Their scheme, called the branch address cache, was a technique for generating multiple fetch addresses each cycle. These multiple addresses index into a multiported (or multi-banked) instruction cache, resulting in several fetch blocks worth of instructions available each cycle. The branch address cache requires instructions to be shifted, aligned, and merged in order to form a packet for issue. Two of the biggest concerns with the Branch Address Cache is its hardware complexity and its amenability to aggressive branch prediction.

Seznec et al. [45] propose a slight, but important, modification to the Branch Address Cache pipeline, thereby enabling the use of more aggressive prediction techniques. Their scheme, called the Multiple-Block Ahead Predictor, associates fetch addresses to target addresses to be fetched n steps ahead, essentially allowing target generation to be pipelined.

For example, consider the dynamic sequence of blocks ABCD. If blocks A and B are being fetched this cycle, then the target entry for block A is associated with next fetch address C, and block B is associated with fetch address D. In the subsequent cycle, blocks C and D are fetched. In both cases, the addresses A and B are associated with targets 2 blocks ahead.

The Collapsing Buffer, proposed by Conte et al. [11] in 1995, boosts fetch rate by addressing a commonly occurring limitation. It collapses taken short forward branches and their targets into a single block. In order for the mechanism to work, the target block must be within the same cache line. Since codes often contain short forward branches, the collapsing buffer is successful at boosting fetch rate.

The trouble with all of the proposed hardware schemes is that they add extra logic in a very critical section of a processor pipeline: between fetch and issue. As will be demonstrated in this dissertation, the trace cache delivers high fetch bandwidth and allows logic complexity to be *moved out* of this critical processing path, and into a section of the pipeline where latency does not affect performance.

3.4.2 Compiler-based techniques

Compiler-based techniques for increasing fetch bandwidth center around increasing sequential runs of instructions by reducing the dynamic occurrence of taken branches. Such techniques were initially developed to increase the scope for compiler-based code optimizations, particularly for VLIW machines, as groups of basic blocks which are likely to be executed together could be optimized as a single unit. These techniques have the side benefit of boosting fetch rate on hardware capable of exploiting sequential runs of instructions.

Trace scheduling [14] is a technique used by the compilers for the Multiflow VLIW machines. With trace scheduling, code is profiled to determine the most frequently executed path, or *trace*, through a portion of a program (usually a subroutine). This trace is treated as a unit, as if all internal branches were removed, giving the compiler a larger scope on which to apply optimizations and scheduling. Fix-up code is added to repair the cases where an internal branch did not behave as expected.

Superblocks [7, 8, 19] build upon the trace scheduling concept by dividing the subprogram along most-likely paths called *superblocks*, each composed of multiple basic blocks. Each superblock has only one entry point, but can have multiple exit points. Superblock

formation allows certain basic blocks to be duplicated in order to enforce the single entry point semantic. Doing so enables aggressive compiler optimization of each superblock.

Hyperblocks [28] are enhancements to superblocks. Hyperblock formation uses predication [21] to increase the flexibility of block formation by allowing blocks to incorporate both paths of statically oscillating branches.

A technique similar to superblock formation was developed by Pettis and Hansen for arranging basic blocks to increase instruction cache efficiency and to reduce the penalty of taken branchs [42]. Since their objective was to increase cache effectiveness and not to increase the compiler's optimization scope (as with superblocks), they were not limited to forming sequential blocks with no side entrances. Calder and Grunwald [4] present a further enhancement to this scheme by eliminating replication outright.

Though these schemes potentially increase fetch rate, their reliance on static predictions for block formation make them vulnerable to run-time variations in branch behavior. The Block-Structured Instruction Set Architecture (BS-ISA) [30], proposed by Melvin and Patt, is an instruction set which is amenable to block-enlargement without a strong reliance on static prediction. Hao et al. [20] demonstrate effective block-enlargement techniques and describe a dynamic prediction scheme for use with the BS-ISA.

In this dissertation, an optimizing production compiler that performs code layout similar to the Pettis and Hansen algorithm is used as a baseline for comparison. The algorithm for block enlargement is slightly different from the one used for superblock formation, as side entrances are not explicitly eliminated, but code replication is minimized. See Section 4.3 for more details.

CHAPTER 4

Experimental Model

4.1 Simulation Environment

The experiments in this dissertation were conducted on a microarchitectural simulator developed using the SimpleScalar 3.0 tool suite [2]. The SimpleScalar 3.0 tool suite provides simulation support for the Alpha AXP ISA [46] via routines for Alpha ISA emulation, system call support, and loader functionality. In addition, SimpleScalar provides general simulation support for cache, memory, and translation lookaside buffer simulation.

The simulator models the pipeline behavior of an out-of-order processor.

The simulator also models off-path behavior. The branch predictor generates predictions and fetch addresses while the processor is both on and off the correct execution path. Thus the path followed by the simulated processor when it is executing off-path instructions is consistent with that of a real processor. The simulator does not simulate system calls. They are however behaviorally emulated using the host machine upon which the simulations are executed.

4.2 Benchmarks and Input Sets

The SPECint95 [51] benchmarks along with five common C applications were used in the experiments performed in this dissertation. The SPECint95 benchmarks are a set of applications representative of a technical workload. Since they are frequently used as a performance metric, they provide a common means of comparison across microarchitectural studies. To supplement the experimental benchmark suite, five UNIX C applications were

added. These additional benchmarks increase the breadth of the benchmark suite, some by adding new types of applications not found in SPEC, others by providing different characteristic behavior (e.g., cache footprints, instruction level parallelism, branch behavior) than the SPECint95 benchmarks. All benchmarks in the experimental benchmark suite, along with a brief description of each, are listed in table 4.1.

Benchmark	Description
compress	Lempel-Ziv based file compression
gcc	Optimizing C Compiler
go	Array-based game playing program
ijpeg	Image compression
li	LISP interpreter
m88ksim	M88100 pipeline simulator
perl	Scripting language interpreter
vortex	Object-oriented database transactor
chess	Chess playing program
ghostscript (gs)	Postscript interpreter
pgp	Encryption program
plot	Function plotting program
simplescalar (ss)	Out-of-order pipeline simulator

Table 4.1: Brief descriptions of each benchmark in the experimental benchmark suite.

The input sets for each benchmark consisted of a training input and a measurement input. The training input was used for profile-based compiler code layout (described in section 4.3) while the measurement input was used to collect the performance data presented in this dissertation. For the SPECint95 benchmarks, whenever possible, a SPEC-provided input set was used. In some cases, however, the SPEC-provided inputs were too long or consumed too much memory for feasible experimentation. Table 4.2 lists all the input sets used in this dissertation. The SPECint95 inputs marked with a dagger (†) are modified. All benchmarks were simulated until completion, except for the benchmarks li and ijpeg on the measurement input sets. These benchmarks were simulated for 500 million instructions.

4.3 Compiler Optimizations

The benchmarks were compiled using the Digital Unix C compiler version 3.5.

Benchmark	Training Set	Measurement Set
compress	35000 e 2231 †	30000 q 2131 †
gcc	ref/dbxout.i	ref/jump.i
go	27 8 train.in †	19 9 train/2stone9.in
ijpeg	train/vigo.ppm	ref/penguin.ppm
li	7queens.lsp †	ref/*.lsp
m88ksim	test/ctl	train/ctl
perl	train/primes.pl	train/scrabbl.pl
vortex	35M †	230M †
chess	train.in	sim.in
gs	graph.ps	sigmetrics94.ps
pgp	tasuki1.jpg	IJPP97.ps
plot	singulr.dem	surface2.dem
ss	-config regress.cfg random	-config default.cfg test-fmath

Table 4.2: The input sets used in this dissertation. All benchmarks, except li and ijpeg, were simulated to completion.

One particular optimization performed by this compiler that is important to instruction fetching is one which arranges basic blocks in the executable such that conditional branches are likely to be not taken. This optimization, along with fetch hardware to exploit it (described in Section 4.6.2), can provide a significant gain in effective fetch rate with low hardware complexity. Because it is a relevant alternative to the trace cache, this combination of hardware mechanism and compiler optimization is one of the fetching schemes investigated in this dissertation.

This optimization of rearranging basic blocks requires profiling in order for the compiler to determine likely outcomes for branches. The block rearrangement algorithm examines a sub-program at a time (a function, for example). Internally, this sub-program is represented as a group of basic blocks connected by edges. Each edge represents a branch outcome. The compiler starts with the most frequent edge, determined by profiling, and then grows a trace forward and backwards from this edge with a mutual most-likely heuristic. It then walks this trace and converts it into an extended basic block by aligning each branch target to make the fall-through the common branch outcome. Blocks at join points may get duplicated. Block duplication is cut off at some threshold of execution frequency. If a trace enters a loop with an average iteration count of two or less, then it will continue copying as it enters the loop and continue around the back edge. This has the effect of peeling off one

or two iterations of the loop to make an extended basic block. If a loop is not peeled and the compiler can find a dominant path through the body, it will make the body a single extended basic block and will typically unroll that loop by four [10].

The first level of block rearrangement is done by the compiler itself. Another level is applied using the OM executable editor. Here, the final executable undergoes another step of block rearrangement. Interprocedural rearrangement occurs within this step. More importantly, blocks within statically-linked library routines and routines in objects not generated by the compiler are also affected. OM uses a variation of the Pettis and Hansen algorithm [42] for block enlargement.

Because of tool problems with using OM, not all benchmarks were fully optimized using both levels of rearrangement. Some benchmarks, while being processed by OM, caused OM to generate error messages and exit. The Table 4.3 lists the maximum level of rearrangement for each benchmark. If both the C compiler (cc) and OM were able to optimize the benchmark, then OM is indicated, whereas if only the compiler was able to optimize, then cc is indicated. Instruction counts while running the measurement input set are also provided.

Benchmark	Rearrangement Level	Instruction Count
compress	OM	119M
gcc	OM	178M
go	OM	145M
ijpeg	OM	500M
li	cc	500M
m88ksim	OM	110M
perl	cc	44M
vortex	cc	242M
chess	OM	243M
gs	cc	235M
pgp	OM	167M
plot	cc	254M
ss	cc	114M

Table 4.3: The maximum rearrangement optimization level of each benchmark along with its instruction count while running the measurement input set.

4.4 Microarchitectural Model

The simulated microarchitecture is a 16-wide implementation of the HPS execution model [39, 40] which performs out-of-order execution using the Tomasulo Algorithm [52]. In the HPS model, instructions are *issued* into a dynamic instruction window (consisting of *node tables*). In the window, instructions wait for their operands to be generated. Once generated, an instruction is ready to be *dispatched* to an execution unit for execution. In this manner, instructions are fetched and issued in program order, but can execute and complete out-of-order.

Effective use of the dynamic instruction window requires that the issuing of instructions be able to slip ahead of their execution. Speculative execution via branch prediction is an essential component of out-of-order execution. In order to recover from incorrect predictions (and exceptions in general), a scheme called Checkpoint/Recovery [22] is used to back up processor state to the point of the misprediction. The checkpointing scheme used in this dissertation creates up to three checkpoints per cycle since up to three conditional branches are fetch per cycle.

In the execution model, instructions undergo six stages of processing: Fetch, Decode, Rename, Window, Execute, Retire. Each stage takes at least one cycle. Each stage is described in the subsections below.

4.4.1 Fetch

The trace cache, icache, branch predictor, branch target buffer, and instruction TLB are all accessed in the fetch stage using a fetch address. The output of this stage is either (1) an aligned packet of up to 16 instructions ready for the decoder, if the packet was fetched from the icache, OR (2) a decoded packet of up to 16 instructions ready for renaming, if the packet was fetched from the trace cache, OR (3) nothing, if both caches miss. Also produced in the fetch stage is a next fetch address for use in the following cycle.

The fetch mechanisms used in this dissertation consists of 128KB of instruction storage and approximately 24KB of branch predictor. In total, five different fetch mechanisms are evaluated, the specifics of each described in the next two sections (Sections 4.5 and 4.6).

For all fetch mechanisms studied, the return address stack (RAS) is fixed at 32 entries and the instruction TLB is fixed at 64 entries. Misses in the Level 1 (L1) instruction cache

initiate a lookup in a 1MB Level 2 (L2) instruction cache with a line size of 128 bytes. The L2 icache latency (from lookup until first line of instructions) is 6 cycles. Misses in the L2 icache access the memory system via a pipelined system bus. Memory accesses take 50 cycles from access until first data, but a new request which matches an in-flight request gets serviced along with the in-flight request, thus does not incur the full memory latency.

If a fetch is serviced by the instruction cache, a Branch Target Buffer (BTB) provides the next fetch address, whereas when a fetch is serviced by the trace cache, the fetch address is selected from the target addresses encoded directly in the trace cache line.

4.4.2 Decode

Recall that instructions are stored in the trace cache in decoded form so only instructions which are fetched from the instruction cache need to be decoded. The process of decoding involves analyzing dependencies across the issue packet (up to 16 instructions). After the dependency analysis, each source operand from each instruction is categorized as either generated by an instruction in a previous issue packet (externally defined) or is generated within the packet (internally defined). Similarly, each destination operand is categorized as persisting beyond the basic block in which is was created or not. The process of decoding 16 instructions from the icache can be possibly overlapped with other operations along the fetch-issue path such as instruction alignment and merging. Also, instructions can be stored in the icache in decoded form, minimizing the amount of processing required after fetch. For these reasons, the decode stage is only modeled as requiring a single cycle.

4.4.3 Rename

When a packet is latched into the rename stage, all source operands are marked indicating whether the operand is externally or internally defined. In the rename stage, all source operands marked as externally generated receive an operand tag via a lookup in the Register Alias Table (RAT). The RAT is a recoverable mapping between architected registers and physical registers [3]. Internal values, on the other hand, require no RAT lookup. Their operand tag can be generated immediately via information encoded within the instruction.

Destination operands which persist beyond their blocks require a physical register and thus require an update to the RAT. Destinations which are overwritten within their blocks, on the other hand, are not allocated a physical register.

The register rename stage was conceptually modeled as ideal: up to 32 sources could be renamed each cycle and up to 16 destinations given physical mappings. Thus, the latency of this stage is always one cycle.

4.4.4 Window

The dynamic instruction window is essentially a large buffer capable of storing 512 in-flight instructions. Instructions are renamed and then issued into the window. They sit in the window monitoring the tags of values generated by executing instructions. Values, when generated, broadcast their unique tags across distribution buses. An instruction is considered ready for scheduling when all its operand values have been generated. Instructions which are ready are scheduled, oldest first, for execution. Up to 16 ready instruction each cycle are dispatched to appropriate functional units.

In HPS terminology, the instruction window is organized into *node tables*. A node table, also called *reservation stations*, stores schedulable operations called *nodes*. With the Alpha ISA, each instruction is considered a node, thus the terms are used interchangeably. The particular implementation used here divides the window into 16 separate node tables, each containing 32 slots for instructions. Each node table is associated with an execution unit, and instructions placed in a particular node table will only execute on the execution unit associated with that table.

Loads and stores have an extra requirement for scheduling beyond those of non-memory instructions. A memory instruction must wait for all previous stores to its address to complete before accessing memory. The memory scheduling algorithm used is perfect memory dependence prediction. Any memory instruction which is dependent on an in-flight store waits for its address, as well as the store's address and data to be generated before it is dispatched to the cache. Any memory instruction which is not dependent on an in-flight store (determined by the perfect detection mechanism) progresses to the caches whenever its address is known. A hardware mechanism which performs this type of memory dependence detection is described in [9].

4.4.5 Execute

Non-memory instructions and the address generation component of memory instructions execute on 16 execution units, each capable of performing all operations. The execution

latencies for instructions is provided in table 4.4.

Operation	Latency
Integer ALU	1
Integer Multiply	3
Integer Divide	20
Address Calculation	1
Floating-point Add	2
Floating-point Multiply	4
Floating-point Divide	12

Table 4.4: The execution latencies of various instructions groups

Up to 10 load or store instructions can be started each cycle, i.e., the Level 1 data cache has 10 ports, each port capable of a read or write. The L1 data cache is 64KB, 4-way set associative, with 32 byte lines. Accesses to the L1 data cache take 1 cycle. Misses are passed to the 1MB L2 data cache and take 6 cycles from request to data. Like with L2 instruction cache misses, L2 data cache misses are passed to the memory system take 50 cycles to complete unless they match an in-flight request.

A value generated by one execution unit is available at the inputs of itself and any of the other 15 in the subsequent cycle. In other words, there is no additional latency for bypassing values between execution units.

A 128 entry 4-way data TLB is used for data accesses.

4.4.6 Retire

The processor is capable of retiring up to 16 instructions each cycle. Retirement happens on the granularity of issue packets when all instructions within an issue packet have completed without generating an exception and predicted branch directions and targets have all been confirmed.

4.4.7 Final note on aggressiveness

The execution model used is similar to the aggressive configuration of the Figures 1.1 and 1.2. This dissertation surveys possible trace cache designs for 5 to 8 years in the future. While a 16-wide superscalar machine with a 128KB 1-cycle instruction cache and 10-ported 64KB data caches with no bypass latency and perfect memory dependence detection

seem unlikely with today's technology, microarchitectural research between now and the time these machines will be feasible to build will have matured possibly beyond what is modeled here. The argument and data presented in Chapter 1 state that gains in execution bandwidth must accompany a corresponding effective fetch rate (via the trace cache) in order for performance to be realized.

4.5 Trace Cache Baseline Configurations

In order to give a broad assessment of trace cache design, three different trace cache mechanisms and two different conventional mechanisms are evaluated throughout this dissertation. The objective here is to show the effects and relative performance of the enhancements presented.

The three trace cache configurations differ in the amount of storage allocated to the trace cache versus the amount allocated to the instruction cache. The first configuration consists of a 128KB trace cache and a 4KB instruction cache. The second configuration consists of equally sized 64KB trace and instruction caches. The last trace cache configuration consists of a 4KB trace cache and a 128KB instruction cache.

For all trace cache configurations, a 24KB multiple predictor described in section 2.4 is used. In the case of a fetch serviced by the instruction cache, a branch target buffer provides the next fetch address.

The instruction cache for these configuration is capable of supplying two adjacent cache lines each cycle, i.e., a fetch to address X will supply both the line containing X and the line containing X+n (where n is the line size). In this manner, packet size is not limited by cache line boundaries. These two fetched lines are aligned and merged so that **up to one fetch block** is supplied each cycle by the icache. This technique of fetching two adjacent lines to form a fetch block is known as split-line fetching [18].

Finally, the baseline trace cache configurations implement a conservative, low-complexity path matching policy: the trace segment selected with the fetch address must exactly match the path selected by the branch predictor, otherwise a trace cache miss is signaled.

4.6 ICache Baseline Configurations

The two conventional mechanisms consist of only instruction caches. They differ in their ability to fetch beyond branches, and subsequently have different branch predictor designs. Both consist of a 128KB instruction cache where two sequential lines are capable of being fetched each cycle.

4.6.1 Single-Block ICache

The Single-Block instruction cache uses the split-line fetch technique to supply up to one fetch block per cycle—at most one branch is fetched per cycle–but no more than 16 instructions. With this technique, the alignment and merge logic is relatively simple. Two ports are required to the instruction cache, but since these requests are to adjacent lines, low-degree interleaving is sufficient. Since only one branch is supplied per cycle, a single branch predictor can be used. The research into single branch predictors is more mature than that of multiple branch predictors, and as a result, their accuracy is higher.

The predictor used in the single block fetch mechanism is a hybrid predictor [6, 12, 29] consisting of a gshare component predictor and a PAs component and is similar to the one used on the Digital/Compaq Alpha 21264 [27]. A selector dynamically determines, per branch, which component predictor is performing better and selects that predictor to supply the prediction for that branch. The accuracy of hybrid predictors has been demonstrated to be higher than equivalently sized single-component predictors. The hybrid predictor used for this configuration is also 24KB: 8KB allocated to the gshare component, 8KB to the PAs component, and 8KB allocated to the selector. In addition to the 24KB in the predictor, the 20KB BTB contains both target addresses and 2K 15-bit per-branch histories used by the PAs predictor. The size of the BTB is chosen to be on-par with the next fetch address information stored on each trace cache line for the trace cache configurations.

4.6.2 Sequential-Block ICache

The Sequential-Block instruction cache allows fetching beyond branches which are predicted not taken. Like the Single-Block Icache, the Sequential-Block Icache uses the split-line scheme. Two adjacent lines are fetched. In this manner, the sequential icache provides up to 16 instructions as long as they are physically sequential in memory. This mechanism

takes advantage of the compiler optimization of block rearrangement to reduce the number of taken branches, increasing the average sequential run length.

In order to accomplish this, however, multiple branches need to be predicted each cycle (in order to determine which are not taken). Thus the predictor also places a limitation in that up to 3 conditional branches can be fetched. The same 24KB multiple branch predictor used by the trace cache configurations is used for this configuration. In addition, a 16KB BTB is used to generate fetch addresses.

Table 4.5 list the five configurations and the basic parameters of each.

Configuration Name	TCache Size	ICache Size	Blocks per Fetch	Br Pred Type	BTB Size
TC.ic	128KB	4KB	3	Multiple	1KB
TC.IC	64KB	64KB	3	Multiple	8KB
tc.IC	4KB	128KB	3	Multiple	16KB
Single	–	128KB	1	Hybrid	20KB
Sequential	–	128KB	3	Multiple	16KB

Table 4.5: The five fetch mechanisms evaluated in this dissertation.

CHAPTER 5

Basic Experiments

The five baseline configurations are evaluated in this chapter. First, the overall performance data in instructions retired per cycle (IPC) is presented for each configuration on the 13 benchmarks. Next, individual factors which affect performance of these configurations are analyzed. The objective of this chapter is to identify the particular weaknesses of the trace cache that motivate the improvements presented in the next chapter.

5.1 Measurements

The performance of the five baseline configurations on the experimental benchmark suite is shown in the two figures below: Figure 5.1 shows performance on the SPECint95 benchmarks and Figure 5.2 shows performance on the additional UNIX applications. The legend for both figures is displayed on the second.

For many benchmarks (gcc, go, li, perl, gs, pgp, plot, ss) the Sequential-Block ICache performs best. The compiler is effectively able to layout the executable, minimizing the occurrences of taken branches. Of the configurations with a trace cache, splitting the instruction storage equally across the trace cache and instruction cache provides the most significant performance. Here, the effect of a high miss rate in the trace cache (which will be discussed in Section 5.2.3) is lessened by the instruction cache, providing a good compromise between high fetch rate and low miss rates. The average performance across all 13 benchmarks is listed in Table 5.1. The Sequential-Block ICache offers a 2% improvement over the best trace cache configuration and a 16% improvement over the Single Icache.

Benchmark	TC.ic	TC.IC	tc.IC	Single	Sequential
Average IPC	4.79	5.08	4.71	4.48	5.18

Table 5.1: Average performance of the baselines on the 13 benchmarks.

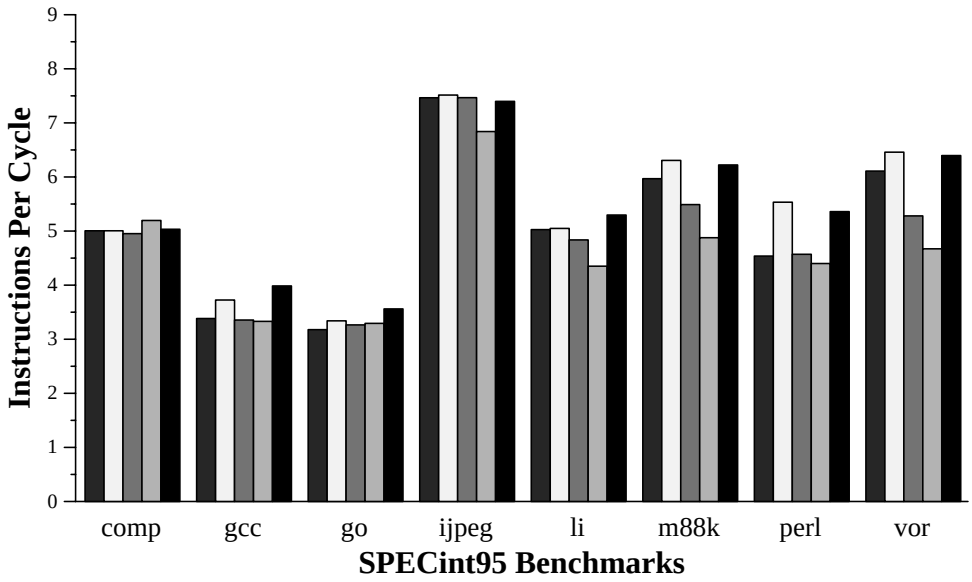

Figure 5.1: The performance in IPC of the five configurations on SPECint95.

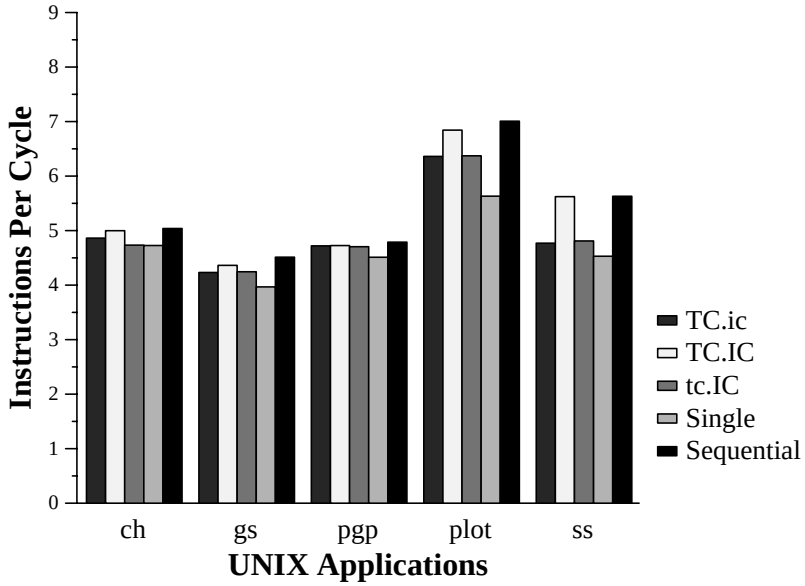

Figure 5.2: Performance of the five configurations on the UNIX applications.

5.2 Analysis

To gain deeper understanding of the phenomena behind these performance trends, one must examine the individual factors which affect performance. Figures 5.3 and 5.4 display a cycle-by-cycle accounting for each benchmark, where each cycle is categorized into one of six categories viewed from the perspective of the fetch mechanism:

1. fetch cycles that result in instructions on the correct execution path (Useful Fetch),
2. fetch cycles that result in instructions off the correct execution path (Branch Misses),
3. fetch cycles that produce no instructions because of a cache miss (Cache Misses),
4. cycles where the machine is stalled due to a full instruction window (Full Window),
5. cycles where the machine is stalled due to trap instructions (Traps), and
6. fetch cycles where the wrong fetch address was generated (Misfetches) [1].

5.2.1 Effective Fetch Rate

The number of cycles required to fetch the program (Useful Fetches) decreases as the effective fetch rate is increased. For most benchmarks, the fetch rate of the Sequential-Block ICache is highest. The average effective fetch rates in instructions per fetch across all the benchmarks are listed in Table 5.2. For the trace cache configurations, the larger the trace cache, the bigger the fetch rate, indicating that a trace cache is able to deliver more instructions per fetch than an instruction cache.

Benchmark	TC.ic	TC.IC	tc.IC	Single	Sequential
Eff Fetch Rate	9.65	9.62	8.44	6.50	9.74

Table 5.2: Average effective fetch rates for the 13 benchmarks.

5.2.2 Branch Misses

For many benchmarks, a significant number of cycles are spent on the wrong execution path because of branch mispredictions. These cycles are labeled Branch Misses in the

[1] Pipeline bubbles due to the different lengths of the trace cache pipeline and icache pipeline are categorized as Misfetches

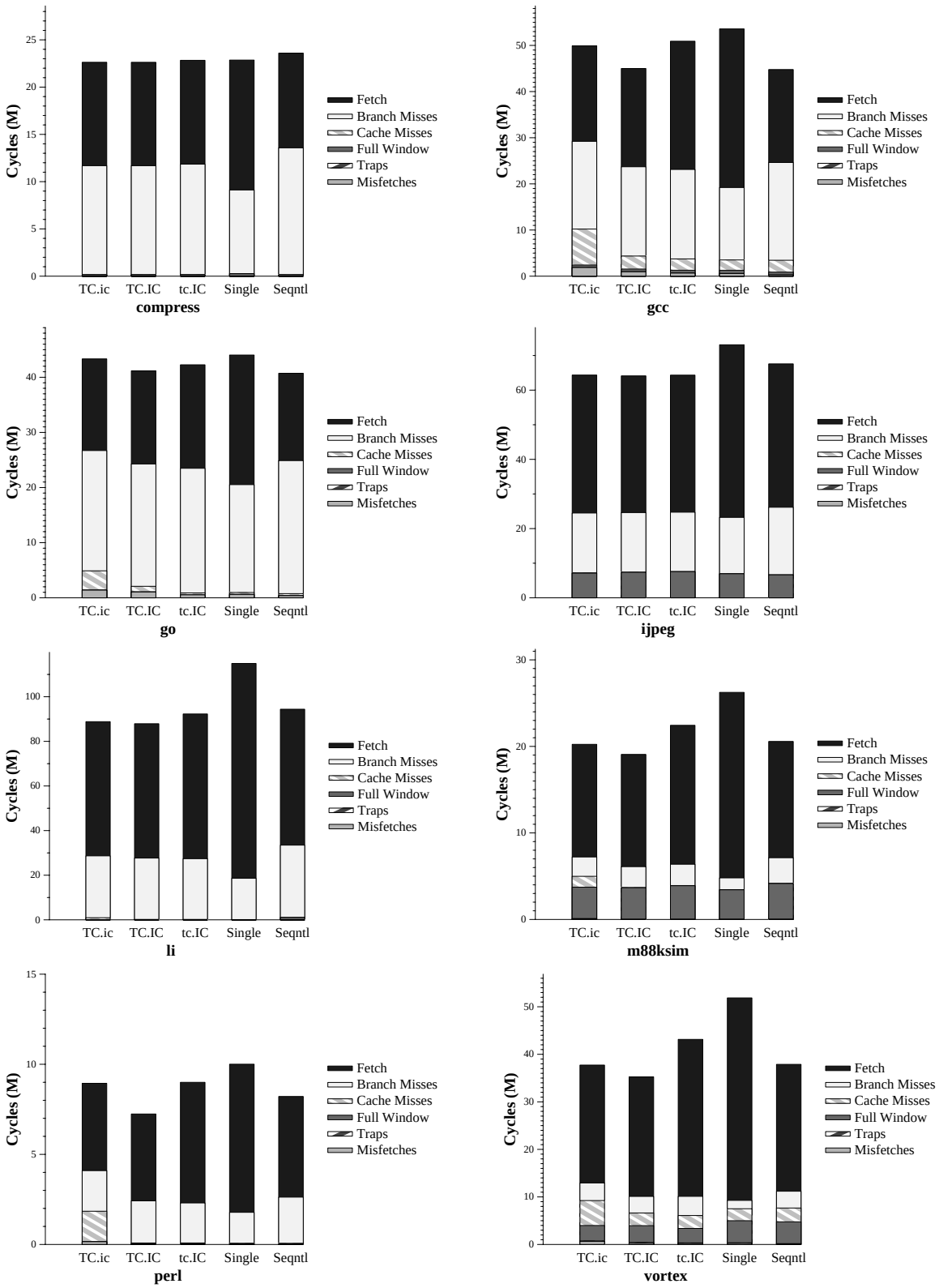

Figure 5.3: Cycle breakdown, SPECint95 benchmarks

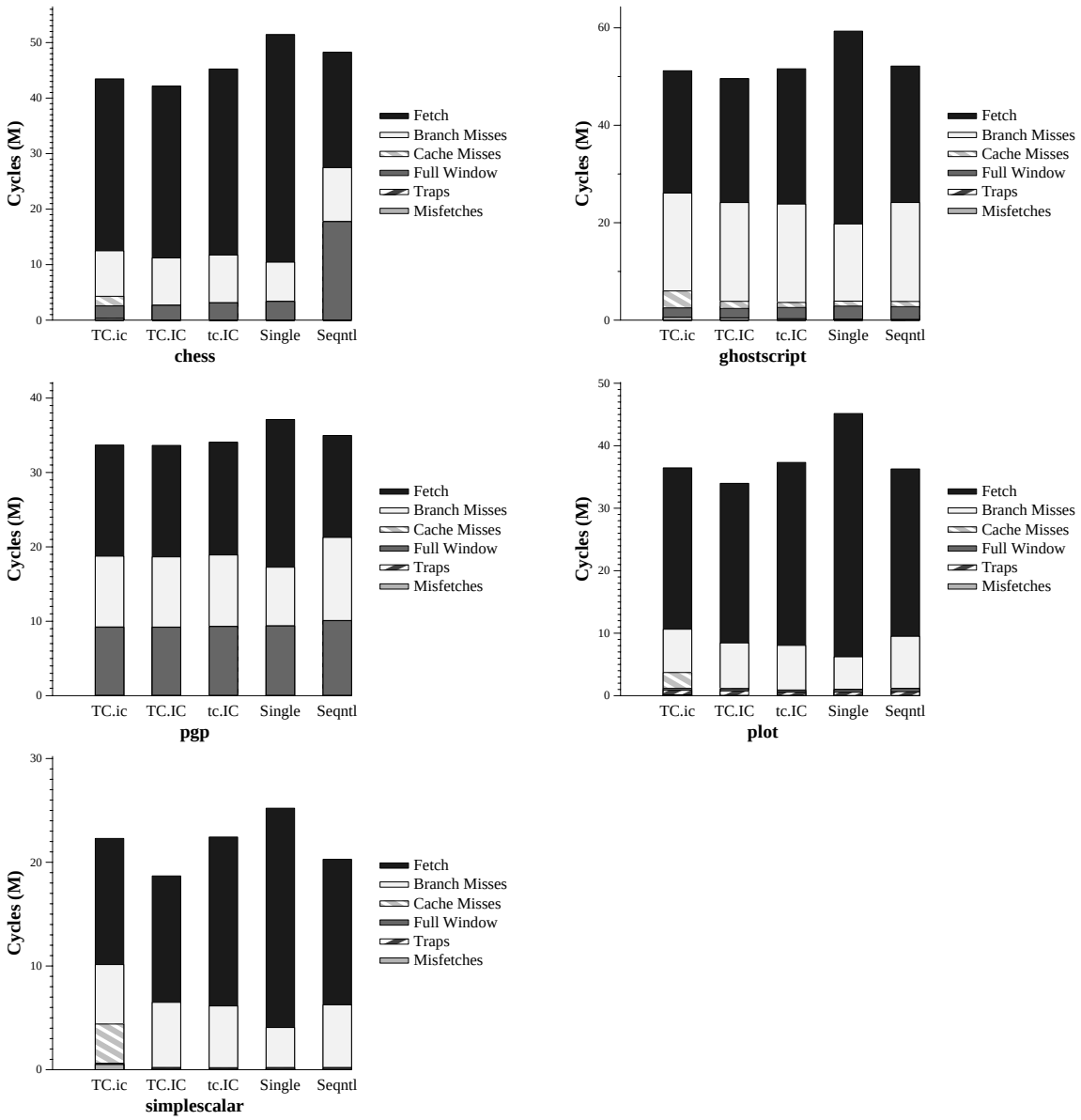

Figure 5.4: Cycle breakdown, UNIX applications

figure. Branch miss cycles are a product of the number of branches mispredicted and the average number of cycles between the prediction of a branch and the eventual decision of its outcome. This time is referred to as the average *branch resolution time*. Because of a phenomenon which will be discussed in Section 8.2, the average branch resolution time grows as the effective fetch rate of the fetch mechanism grows.

Note that absolute number of cycles of Branch Misses is lower for the Single-Block ICache configurations on almost all benchmarks. The hybrid predictor used in the Single-Block ICache is an advanced prediction scheme that is more accurate than the multiple branch

predictor used for the other configurations. The multiple branch predictor is geared towards predicting multiple branches per cycle whereas the hybrid is not. The multiple branch predictor suffers degradation in performance because branch information is inefficiently stored within its structures to accommodate the multiple prediction capability. The branch misprediction rates for the individual benchmarks and configurations are listed in Table 5.3.

Benchmark	TC.ic	TC.IC	tc.IC	Single	Seqntl
compress	8.33	8.33	8.34	5.00	8.49
gcc	7.61	7.69	8.03	5.98	7.39
go	17.09	17.07	17.33	14.29	16.32
ijpeg	9.33	9.28	9.33	8.46	9.14
li	4.20	4.20	4.20	2.72	4.26
m88ksim	1.89	1.89	2.02	0.79	2.00
perl	2.66	2.60	2.80	1.24	2.76
vortex	1.65	1.52	1.92	0.74	1.34
chess	2.33	2.32	2.42	1.83	2.25
gs	5.49	5.53	5.64	4.16	5.44
pgp	5.26	5.24	5.27	4.35	5.27
plot	2.59	2.60	2.66	1.53	2.63
ss	4.70	4.72	4.93	3.36	4.61
Average	5.63	5.61	5.76	4.19	5.53

Table 5.3: The conditional branch misprediction rates (in percentage) of the five baseline configurations.

The average resolution time, in cycles, for a mispredicted branch is listed in Table 5.4. There is a complex interaction between two main factors affecting resolution time: pipeline length and fetch rate. First, branch resolution time increases as more fetches are supplied from the instruction cache. Recall that the icache path (even with the trace cache) requires an additional pipeline stage to accommodate the alignment and decoding required to form a packet. Any branch fetched from the icache will require an extra cycle for resolution. Therefore, the icache-based schemes suffer from higher resolution times. Second, as mentioned earlier, resolution times increase as the effective fetch rate is increased. The phenomenon behind this is subtle and will be examined more thoroughly in Section 8.2—basically, as fetch rate is increased, more instructions upon which the branch is dependent are fetched along with the branch, thus the branch waits longer in the instruction window before it executes. Since the Sequential-Block ICache and the TC.IC configurations have high fetch

rates and a high number of fetches from the icache, they suffer from higher resolution times.

Finally, notice from Table 5.3 that the Sequential-Block ICache suffers from roughly the same misprediction rate as the trace cache configurations. Also, notice in Table 5.4 that it suffers from the highest resolution times. These two factors account for the large branch miss penalties associated with this configuration (see Figures 5.3 and 5.4).

Benchmark	TC.ic	TC.IC	tc.IC	Single	Sequential
Ave mispredicted brn resolution time	8.82	9.07	8.83	9.53	10.40

Table 5.4: Average branch resolution time, in cycles, for mispredicted branches.

5.2.3 Trace Cache Misses

The fetch rate for the trace cache is highly dependent on frequently hitting in the trace cache. The trace cache miss rates in misses per 1000 instructions for the three trace cache configurations are listed in the left portion of Table 5.5. In the right portion, the overall misses per 1000 instructions for the entire fetch mechanism (i.e., misses in both the trace cache and the instruction cache) are listed. The average for each configuration is provided to identify the general trends in miss rate. A trend of importance is that the trace cache miss rate remains high, even if the trace cache is large. In order for an increase in effective fetch rate to be realized, a significant number of fetches must come from the trace cache. Therefore, a high trace cache miss rate is not acceptable.

The trace cache involves trading off efficient caching of the executable for caching it in a manner suited for wide fetch. Instructions can be cached multiple times in the trace cache; this duplicate storage can degrade cache hit rates. The effects of duplication and techniques to control it will be discussed in Section 8.3. Primarily due to duplication, an instruction cache will suffer from fewer misses than a trace cache of comparable size.

However, for some benchmarks (compress, ijpeg, pgp), the trace cache miss rate does not change significantly as the size of the trace cache is increased. These benchmarks have small instruction working sets (compress can effectively be cached in 4KB) so it is expected that these benchmarks should not suffer significantly from misses since they will not be

affected by any lack of space in a 128KB trace cache.

Since capacity misses can be ruled out for these benchmarks, two other sources of misses can account for this behavior. First, conflict misses could be a factor. Even though the trace and instruction caches were modeled as 4-way associative, it could be that the trace cache indexing scheme creates many hot spots and thus requires a higher degree of associativity to be effective. This factor is examined in 7.1. Second, hits to the trace cache are signaled only if both the tag matches the tag portion of the fetch request AND the path predicted by the branch predictor completely matches the path encoded within the trace segment. Recall that the baseline trace cache configurations do not implement Partial Matching.

Benchmark	TC.ic	TC.IC	tc.IC	TC.ic	TC.IC	tc.IC	Single	Seqntl
compress	16.80	16.80	17.62	0.02	0.00	0.00	0.00	0.00
gcc	37.28	43.49	113.03	14.90	2.18	1.00	0.97	0.95
go	46.24	50.53	81.80	11.14	3.81	1.03	0.97	1.04
ijpeg	11.25	10.00	16.39	0.04	0.00	0.00	0.00	0.00
li	54.58	52.37	71.96	0.70	0.00	0.00	0.00	0.00
m88ksim	16.02	15.97	70.55	4.25	0.01	0.00	0.00	0.00
perl	34.58	33.62	112.48	16.38	0.06	0.04	0.03	0.04
vortex	23.68	28.13	93.33	5.85	0.93	0.45	0.45	0.43
chess	75.31	75.13	100.27	2.91	0.03	0.00	0.00	0.00
gs	17.08	38.05	57.94	4.99	1.45	0.53	0.52	0.51
pgp	35.93	17.37	17.93	0.06	0.01	0.00	0.00	0.01
plot	25.04	23.26	56.13	4.17	0.03	0.01	0.01	0.01
ss	27.49	27.88	98.87	14.35	0.05	0.02	0.01	0.02
Average	33.69	34.08	63.21	4.15	0.53	0.20	0.19	0.19

Table 5.5: The left half of this table lists the trace cache misses per 1000 instructions retired. The right half lists icache misses (for the trace cache configurations, this implies a tcache miss as well) per 1000 instructions retired.

5.3 Conclusions

The preliminary performance data indicate that the Sequential-Block instruction cache works best amongst the five baseline configurations. This configuration delivers the highest fetch rate coupled with low icache miss rates, but suffers from a larger loss due to branch misses.

The trace cache configurations suffer severely from trace cache misses. The TC.ic (128KB trace cache, 4KB icache) configuration suffered an average of 33.69 trace cache misses in delivering 1000 instructions. Of the trace cache configurations, the TC.IC (64KB trace cache, 64KB icache) configuration had the highest overall performance because it coupled the high fetch rate possible with the trace cache with the high hit rate possible with an icache.

CHAPTER 6

Enhancements

6.1 Overview

In the previous chapter, it was noted that trace cache performance was severely degraded by trace cache misses. Furthermore, since it requires several branch predictions per cycle, the trace cache suffered from a high penalty due to conditional branch mispredictions. In this chapter, several enhancements are proposed to address both of these concerns. Partial Matching increases the trace cache hit rate by allowing the trace cache to signal a hit even when only a portion of the selected path matches the trace segment. Inactive Issue allows the trace cache to hedge against a branch misprediction by allowing issue of the path of the branch that was not selected by the predictor.

Performance of the trace cache can also be enhanced by making it more effective at delivering instructions. The two final enhancements proposed in this chapter directly boost the effective fetch rate. Branch Promotion converts highly biased branches into assertion-like instructions which act like unconditional branches. Trace Packing relaxes the atomic treatment of basic blocks during segment creation.

All the enhancements, directly or indirectly, increase the effective fetch rate of the trace cache. The primarily role of the trace cache is to boost the fetch rate without compromising the hit rate or the branch prediction rate, thereby achieving an overall increase in instruction fetch bandwidth.

6.2 Partial Matching

As noted in the previous section, a large fraction of trace cache misses may be due to the fact that the entire predicted path must match the selected trace segment in order to signal a trace cache hit. For this experiment, a new hit strategy called *Partial Matching* is evaluated. With Partial Matching, the predictor selects which blocks encoded on the matching trace cache line are fetched and which are discarded.

Figure 6.1 (which is the same as Figure 2.2) demonstrates Partial Matching. A request is made for block A. The trace cache responds with ABC. The predictor predicts the control flow to go from A to B to D. With Partial Matching, the partial segment AB is supplied.

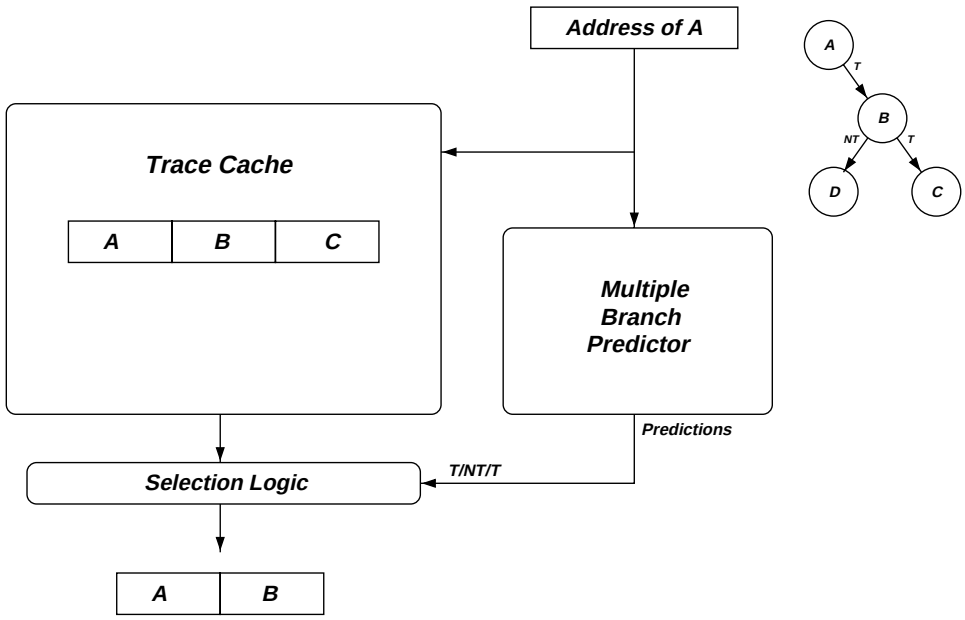

Figure 6.1: The trace cache and branch predictor are indexed with the address of block A. The inset figure shows the control flow from block A. The predictor selects the sequence ABD. The trace cache only contains ABC. AB is supplied.

6.2.1 Implementation issues

Partial Matching comes at a cost, however. To implement Partial Matching as simulated here, four target addresses must be stored in each trace cache entry. Figure 6.2 shows the possible next fetch addresses with and without Partial Matching. With no Partial Matching, only two possible targets exist for each trace segment. Selection between four as opposed

to two will likely increase the critical access path of the fetch stage. The selection of the next target address is essential to complete in one cycle; the selection of instructions can be pushed into the next cycle as unwanted blocks can be invalidated as the instructions within the packet are renamed.

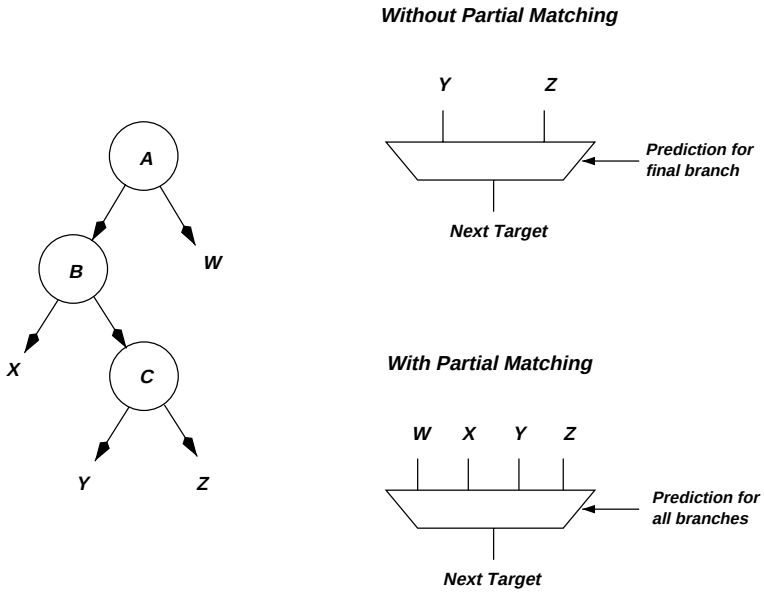

Figure 6.2: Part of the implementation cost of Partial Matching is the selection between four target address as opposed to two.

Several options exist for dealing with this increase in implementation cost. If cycle time is the concern, then two of the four targets can be preselected as most likely and the branch predictor can select between these two. If the real target is one of the other two, then a misfetch penalty is suffered. If storage costs are a concern, then the target storage can be decoupled from the trace cache storage. This separate trace target buffer is more adaptive and can take advantage of the property that many trace segments will likely require fewer than all four possible targets. A miss in the trace target buffer will incur the same small misfetch penalty as picking an incorrect target. Finally, a coupled predictor such as the Next Trace Predictor [24] can be used. With this scheme, the pattern history table of the branch predictor is replaced with a table containing target addresses. Instead of predicting individual branches, and then using these predictions to select a target, the process is coupled into a single table lookup. The branch (or path, as Jacobsen et al. propose) history indexes into a table which then provides the next target to fetch from. With this scheme, target addresses which are never accessed are never allocated storage.

6.2.2 Measurements

The data plotted in Figures 6.3–6.5 show the implications on performance of Partial Matching versus requiring a full match. Each figure represents a different baseline trace cache configuration. In the last figure in this sequence, Figure 6.6, the three configurations with Partial Matching are plotted on the same graph. With the addition of Partial Matching, the large trace cache, small icache configuration TC.ic performs about the same as an evenly split TC.IC configuration. The average IPC for TC.ic is 5.61, TC.IC is 5.59, and tc.IC is 4.95.

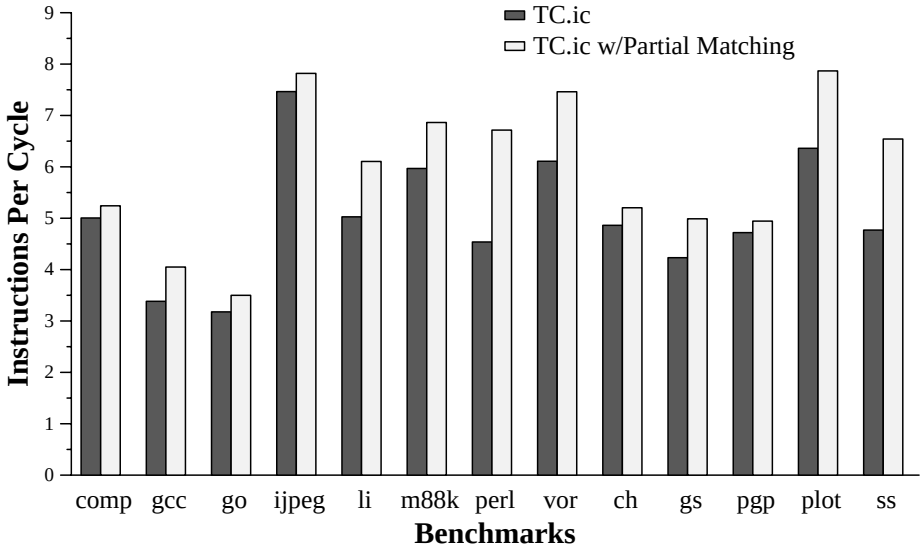

Figure 6.3: The TC.ic configuration with and without Partial Matching.

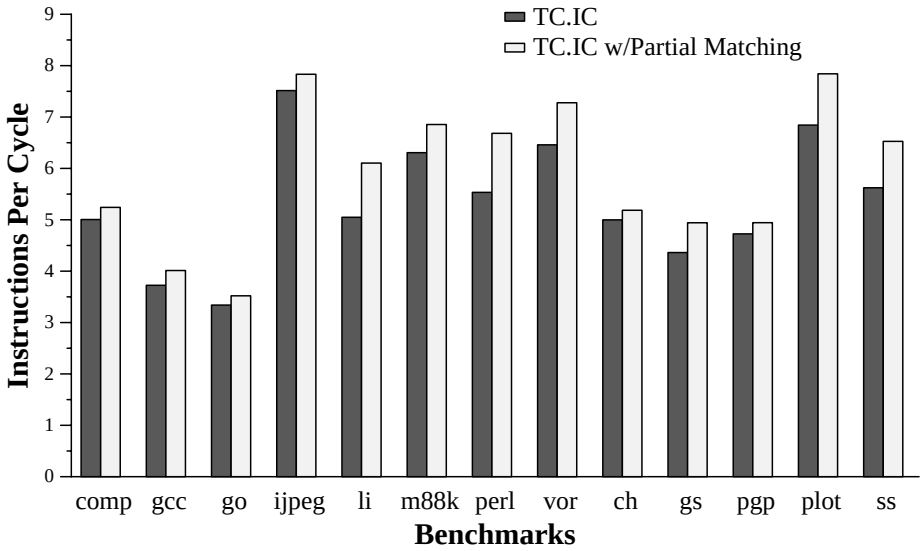

Figure 6.4: The TC.IC configuration with and without Partial Matching.

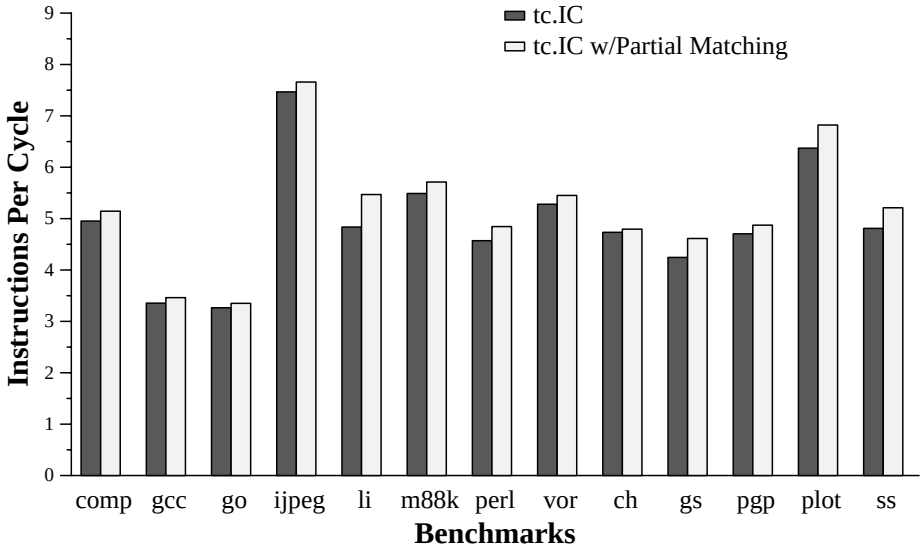

Figure 6.5: The tc.IC configuration with and without Partial Matching.

6.2.3 Analysis

Table 6.1 lists the trace cache misses per 1000 instructions for the three configurations with Partial Matching added. The average trace cache miss rate for the TC.ic configuration drops from 33.69 misses every 1000 instructions without Partial Matching (as reported in Table 5.5) to 1.52 misses every 1000 instructions. The additional flexibility of Partial Matching effectively addresses the trace cache miss problem observed with the baseline

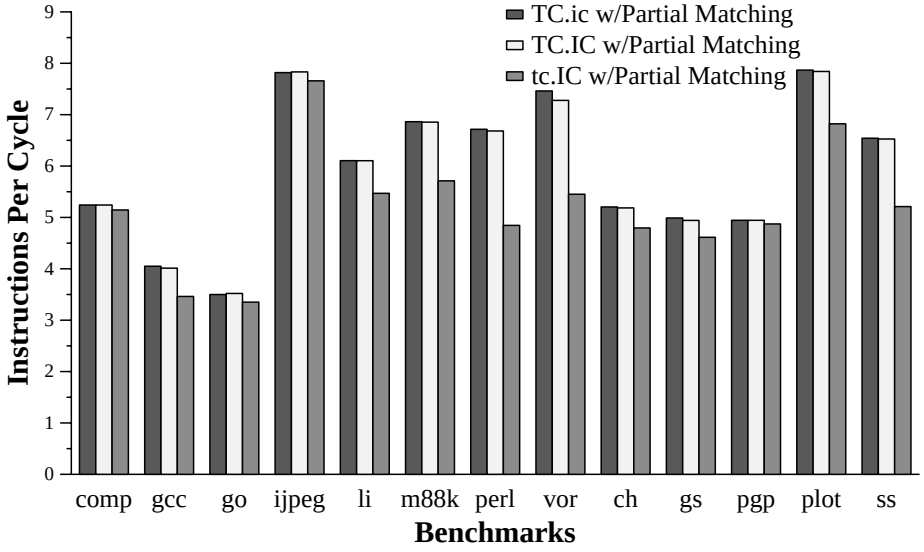

Figure 6.6: All three trace cache configurations with Partial Matching. The TC.ic configuration performs slightly better than the evenly split TC.IC configuration.

configuration in Section 5.1.

Benchmark	TC.ic	TC.IC	tc.IC
compress	0.01	0.01	4.64
gcc	6.56	14.51	96.92
go	15.84	22.14	63.43
ijpeg	0.01	0.01	10.15
li	0.02	0.08	28.29
m88ksim	0.04	0.30	59.28
perl	0.14	0.83	94.59
vortex	1.19	4.81	83.69
chess	0.19	1.14	31.93
gs	1.65	4.78	33.32
pgp	0.04	0.05	5.84
plot	0.43	0.88	41.92
ss	0.39	1.74	81.70
Average	1.52	3.11	40.01

Table 6.1: The trace cache miss rates in misses per 1000 instructions are displayed for the three configurations with Partial Matching added.

Partial Matching also boosts the effective fetch rate of these configurations. For the fetch rate to go up, the number of instructions fetched from the trace cache on a partially matched hit must be larger on average than the number which could be delivered from the icache, i.e., larger than one fetch block's worth. The effective fetch rate of the TC.ic configuration with Partial Matching is 11.20 instruction per fetch as opposed to 9.62 for the same configuration without Partial Matching. So even on partially matched hits, the trace cache often delivers more than a single block.

6.3 Path Associativity

Path Associativity relaxes the constraint that different segments starting from the same fetch block cannot be stored in the trace cache at the same time. Path Associativity allows segments ABC and ABD to reside concurrently in the cache whereas a non-path-associative trace cache allows only one segment starting at A to be resident in the trace cache at any instant in time.

6.3.1 Implementation issues

A path associative trace cache datapath is shown in figure 6.7.

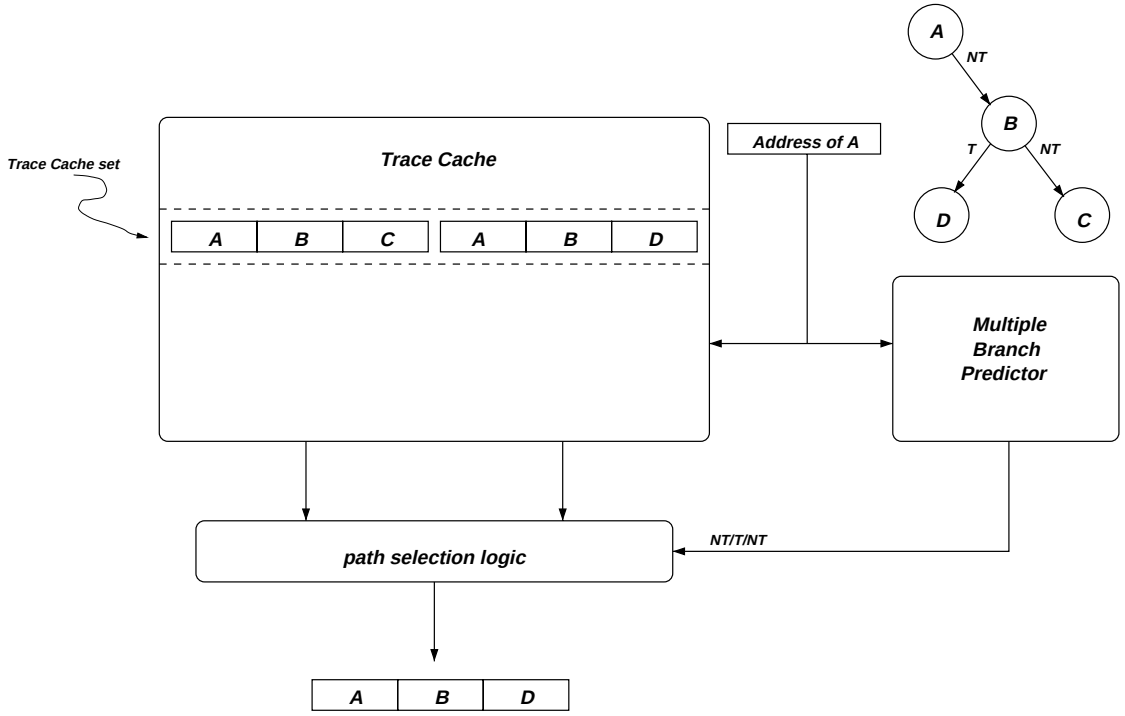

Figure 6.7: The trace cache drives out all segments in the set. The prediction is used to select the longest matching segment.

All segments starting at the same address are stored in the same set of the cache. In the same manner as the non-path-associative case, the fetch address is used to index into the trace cache and a tag match is performed to find the matching line. The tag matching processes of a path-associative cache and a non-path-associative cache have a slight but crucial difference. Both require a tag match of the upper bits of address and both require using the branch prediction to select the proper next fetch address. However, the path associative cache requires that the longest matching path be determined before the matching line is selected. Finding the longest matching path requires completing the address match first. Only after the address match is complete, can the longest matching path and thus the next fetch address and instructions to supply be selected. It is likely that the line selection time for the path associative cache will be longer, possibly impacting cycle time.

6.3.2 Measurement

The data plotted in Figure 6.8 indicate that Path Associativity has little effect on the performance of the baseline trace cache. Adding Path Associativity increases the number of segments that map into a particular set; thus additional misses may occur due to increased set conflicts. This effect can be seen for the benchmarks li and vortex, where enabling path associativity causes performance to drop slightly. Therefore the experiment was conducted on a path-associative trace cache with set-associativity of 4 and of 8. In both cases, the performance gain from Path Associativity is very small.

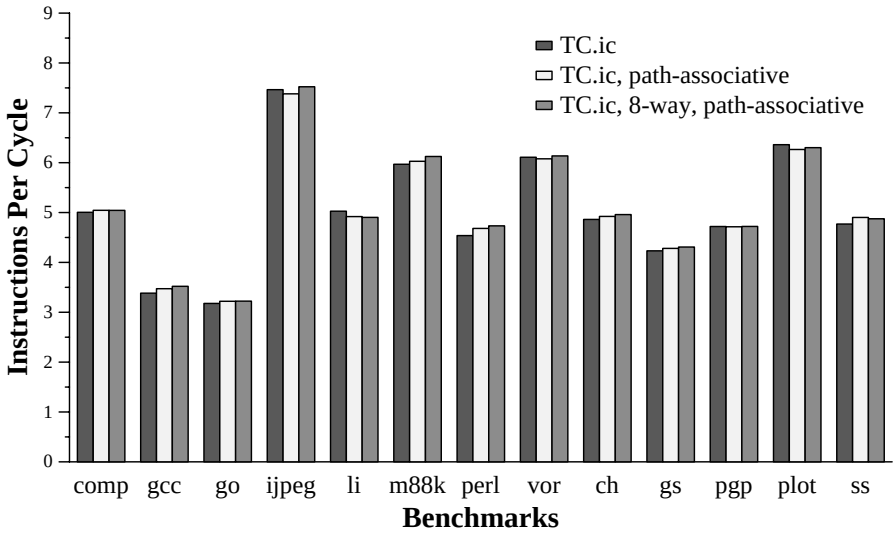

Figure 6.8: Performance of Path Associativity on the TC.ic configuration.

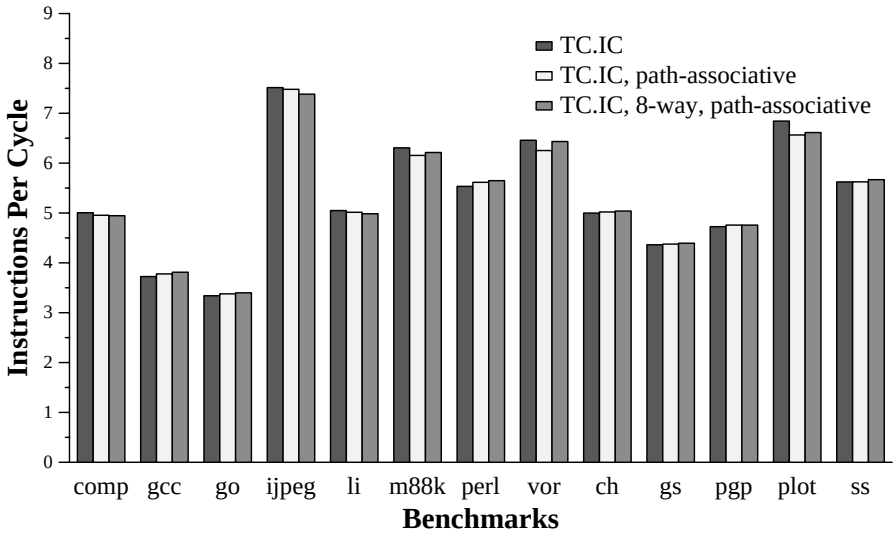

Figure 6.9: Performance of Path Associativity on the TC.IC configuration.

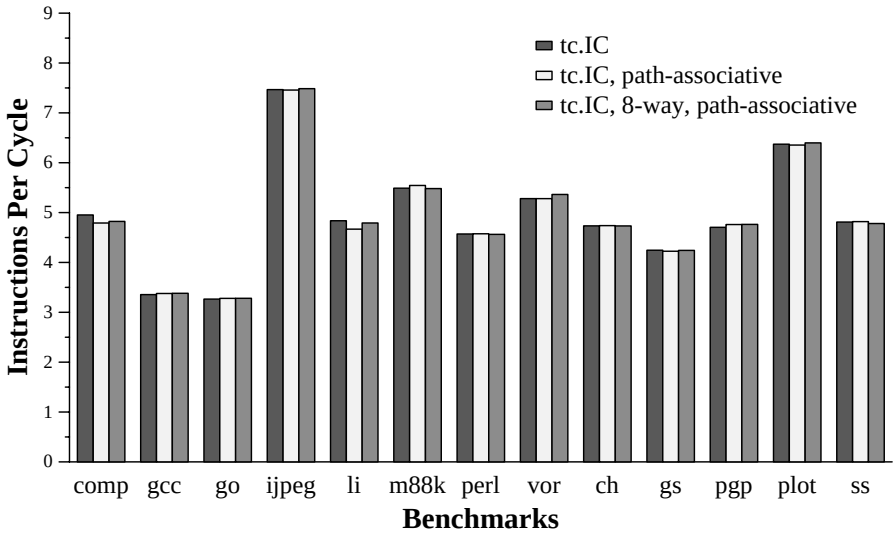

Figure 6.10: Performance of Path Associativity on the tc.IC configuration.

6.3.3 Analysis

Partial Matching and Path Associativity are different approaches for increasing performance by decreasing the trace cache miss rate. The performance data from the previous section suggest that Path Associativity does a rather poor job at it. The trace cache miss data (in misses per 1000 instructions) presented in Table 6.2 support this observation. Even though the caching policy is made more flexible with Path Associativity, allowing better adaptability to dynamic program behavior, the overall trace cache miss rates are only slightly lower than the trace cache miss rates of the baseline configurations (provided in the last row). The resulting average effective fetch rates are 9.56 for the TC.ic configuration, 9.72 for the TC.IC configuration, and 8.48 for tc.IC.

Though Path Associativity has the potential to increase performance, several factors combine to reduce its effectiveness. Path Associativity increases the number of items which map into the trace cache. So for a trace cache of a particular size, the number of capacity misses increases when Path Associativity is added. Since all these new items all map to the same set as the corresponding original items, the number of conflict misses also increases. So in order for Path Associativity to be a performance win, both the size and set-associativity of the trace cache must be increased. Furthermore, Path Associativity addresses trace segments which oscillate between two or more paths. The further these oscillations are spaced apart in time, then the less effective Path Associativity becomes. Partial Matching is a better policy for dealing with these oscillating trace segments.

Benchmark	TC.ic	TC.IC	tc.IC
compress	11.39	12.22	17.51
gcc	28.94	37.13	111.56
go	37.58	42.78	79.09
ijpeg	10.51	9.02	15.67
li	49.98	49.08	78.90
m88ksim	11.06	15.79	65.17
perl	27.54	27.60	112.03
vortex	19.62	26.34	93.18
chess	30.01	10.43	58.07
gs	28.14	30.68	56.99
pgp	20.25	19.78	19.73
plot	24.06	23.88	55.24
ss	22.18	24.06	98.30
Average	26.32	26.06	60.11
Ave w/o Path Assoc	33.69	34.08	63.21

Table 6.2: The trace cache miss rates in misses per 1000 instructions with Path Associativity added.

6.4 Inactive Issue

Partial Matching increases the number of instructions that are issued each cycle but it does not take advantage of the entire segment of instructions fetched from the trace cache. Blocks within the trace cache segment which do not match the predicted path are discarded. As long as the prediction is correct, this does not impact the effective fetch rate. If the prediction is incorrect however, an opportunity to issue a greater number of correct instructions has been missed.

With Inactive Issue [16] all of the blocks within a trace cache line are issued into the processor whether or not they match the predicted path. The blocks that do not match the prediction are said to be *issued inactively*. Although these inactive instructions are renamed and receive physical registers for their destination values, the changes they make to the register mapping (i.e., the Register Alias Table) are not considered valid for subsequent cycles. Thus instructions along the predicted path view the speculative state of the processor exactly as if the inactive blocks had not been issued. When the branch that ended the last active block resolves, if the prediction was correct, the inactive instructions are discarded and their physical registers deallocated. If the prediction was incorrect, the processor has

already fetched, issued, and possibly executed some instructions along the correct path.

This technique has a few advantages. Primarily, it reduces the impact of branch mispredictions. Inactive Issue is able to hide a portion of the branch resolution latency by making some correct path instructions (which follow a mispredicted branch) available for processing earlier. When the mispredicted branch resolves, the recovery state of the processor is further along the correct path than it would have been if the inactive instructions had not been issued. Furthermore, if any inactively issued branches have resolved, the predictions made for them are not used. Their actual outcomes are known.

Inactive Issue also has the potential to reduce the amount of block duplication within the trace cache. Without Inactive Issue, when a mispredicted branch resolves, the next fetch might correspond to an address that is already within the trace cache but is not accessible, as it could be an interior block (i.e., blocks B and C of the segment ABC.) With Inactive Issue however, the block no longer has to be refetched, and need not be rewritten to the trace cache into a new entry (though is may be already duplicated for other reasons). Block duplication is therefore reduced and the trace cache storage is used more effectively. This effect also reduces the pressure on the multiple branch predictor, causing a reduction in interference.

Figure 6.11 illustrates the blocks issued from a fetched cache line by the three different policies: no Partial Matching, Partial Matching, and Inactive Issue.

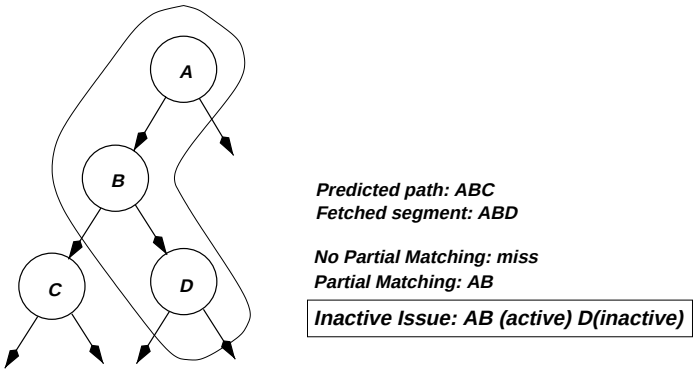

Figure 6.11: An example of the different issue policies.

6.4.1 Implementation issues

To implement Inactive Issue, modifications must be made to the renaming and recovery structures. The HPS execution model uses a checkpointed register alias table to maintain

both the architectural and speculative state of the processor. The changes needed to implement Inactive Issue include adding an active bit to each checkpoint in the table. As the checkpoints are created, this bit is set if the instructions in the corresponding block are issued actively and the bit is cleared if the instructions are issued inactively. The most recent active checkpoint is used as the speculative state of the machine when new instructions are issued. When a branch resolves and is determined to be mispredicted, the inactive checkpoints immediately following the resolved checkpoint become active and all subsequent checkpoints, corresponding to instructions along the incorrect path, are flushed from the pipeline and the instruction window. The fetch proceeds from the (possibly predicted, possibly actual) target of the newly activated checkpoint. If the branch resolves correctly predicted, the inactive checkpoints are simply invalidated.

6.4.2 Measurement

Figures 6.12–6.14 present the performance benefits of Inactive Issue. Since Inactive Issue is a further enhancement to Partial Matching, the Partial Matching results from Section 6.2 are provided for reference. Inactive Issue is a hedge against mispredicted branches, therefore the value of Inactive Issue (over simple Partial Matching) is greater when the branch predictor performs poorly; Inactive Issue is more helpful for programs with a higher misprediction rate as demonstrated by the boost in performance on benchmarks gcc and go. On the TC.ic configuration, the performance of gcc is improved by 4% and the performance of go by 7% over that of Partial Matching. The performance differential, in general, between Inactive Issue and Partial Matching would be expected to increase if a smaller, less effective branch predictor were used.

6.4.3 Analysis

The cycle breakdown for the benchmarks gcc and go on the TC.ic configuration is presented in Figure 6.15. In this figure, the baseline TC.ic is compared with TC.ic plus Partial Matching and TC.ic plus Inactive Issue. Only gcc and go are shown because of their high branch misprediction rates. The main feature to observe in this figure is that Inactive Issue reduces the number of cycles lost due to branch mispredictions for both benchmarks.

While Inactive Issue reduces the number of cycles to resolve a branch by making correct-path instructions following a mispredicted branch available before the resolution of that

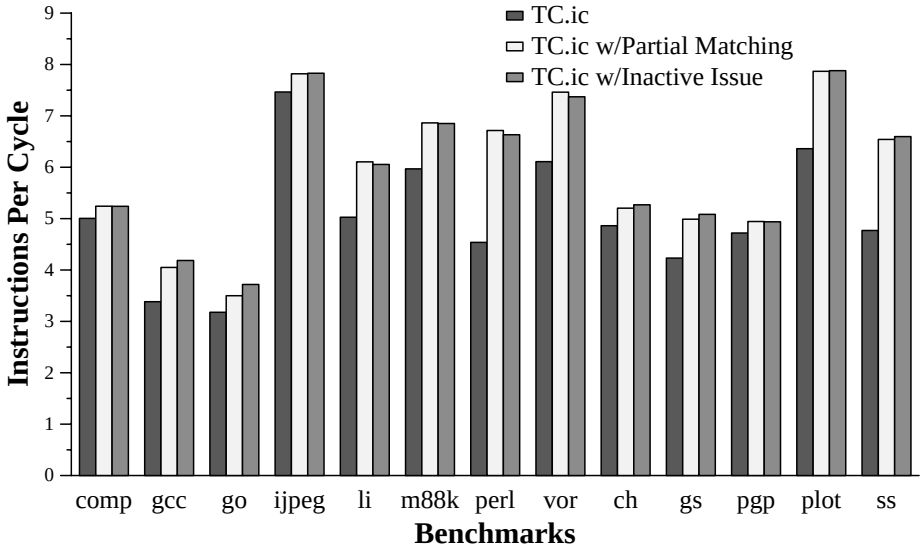

Figure 6.12: Performance of Inactive Issue on the TC.ic configuration. The configuration with Partial Matching is shown for contrast.

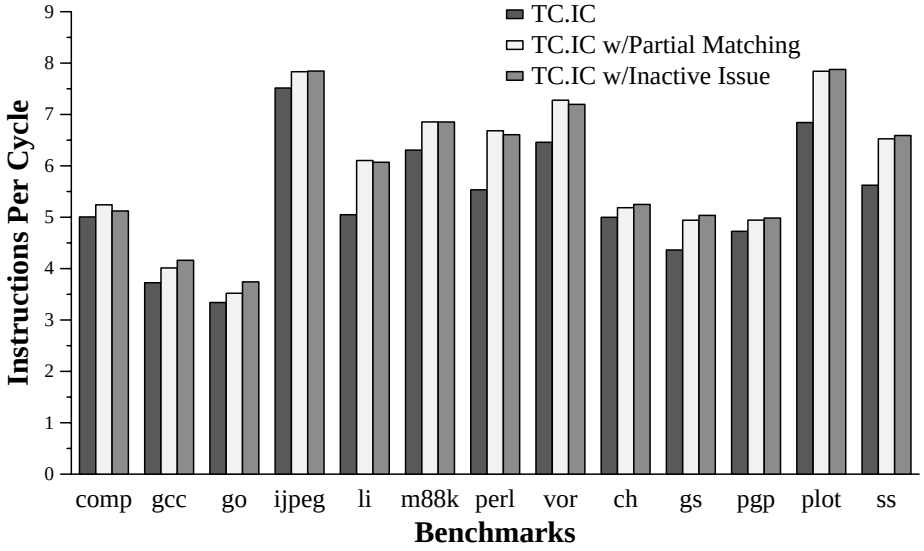

Figure 6.13: Performance of Inactive Issue on the TC.IC configuration. The configuration with Partial Matching is shown for contrast.

branch, there is a negative factor which causes resolution time to increase. Inactive Issue increases the number of instructions competing for resources at any time. Many instructions issued by this policy will eventually be discarded (i.e., if the corresponding branch is correctly predicted) but they still compete with real instructions for functional units and memory ports. The loss in performance observed on some benchmarks, such as perl, is due to this effect. Table 6.3 below quantifies the average number of cycles that ready instruc-

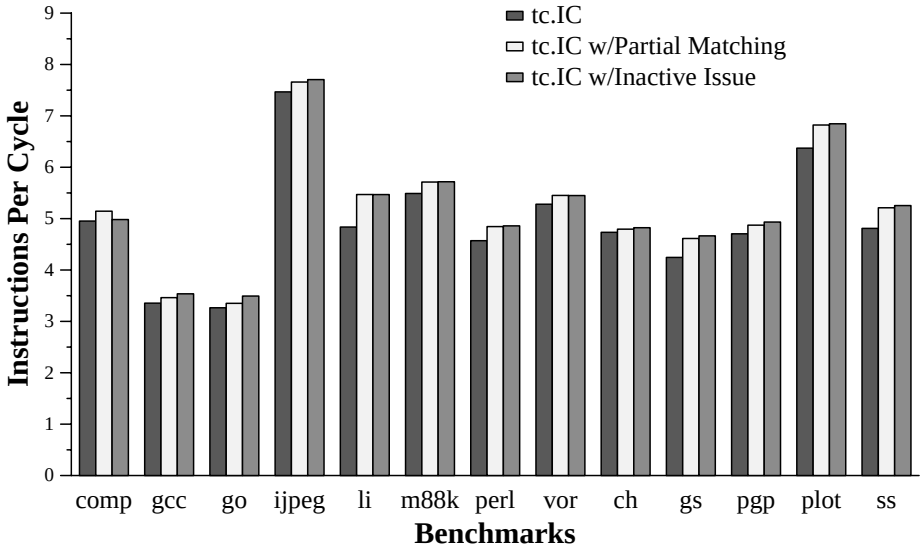

Figure 6.14: Performance of Inactive Issue on the tc.IC configuration. The configuration with Partial Matching is shown for contrast.

tions wait for functional units with Partial Matching compared to with Inactive Issue for the TC.ic configuration, broken down per benchmark. While this absolute number of cycles is fairly small, this increase accumulates across all instructions in a dependency chain, potentially increasing branch resolution time.

Benchmark	Partial Matching	Inactive Issue
compress	0.13	0.14
gcc	0.13	0.14
go	0.08	0.10
ijpeg	0.48	0.50
li	0.16	0.19
m88ksim	0.24	0.26
perl	0.22	0.24
vortex	0.25	0.26
chess	0.09	0.10
gs	0.18	0.18
pgp	0.19	0.21
plot	0.34	0.37
ss	0.26	0.27

Table 6.3: The average number of cycles a ready instruction waits for a functional unit.

To deal with the effect of inactive instructions competing for resources with active

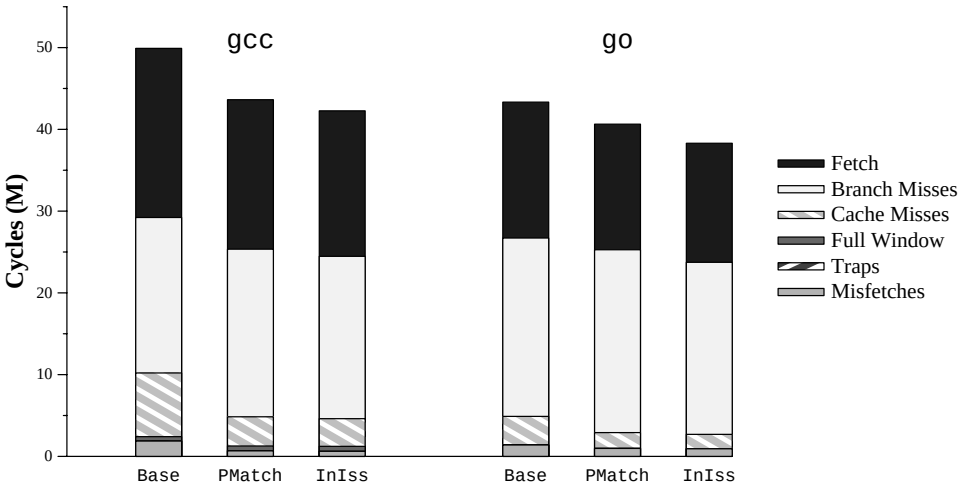

Figure 6.15: Cycle breakdown for the TC.ic configuration, without Partial Matching, with Partial Matching, and with Inactive Issue.

instructions, the scheduling policy was modified to always give priority to active instructions. With this policy, if an active instruction and an inactive instruction are both requesting the same execution unit, then the active instruction will win, regardless of whether the inactive instruction is older. Figure 6.16 compares the performance of the TC.ic configuration with Partial Matching, with Inactive Issue, and with Inactive Issue with the modified scheduling policy. The benchmarks where lots of instructions are issued inactively (gcc and go) benefit slightly with this policy. The percentages over the bars indicate the performance difference between Partial Matching and Inactive Issue with the new policy. None of the benchmarks drop in performance.

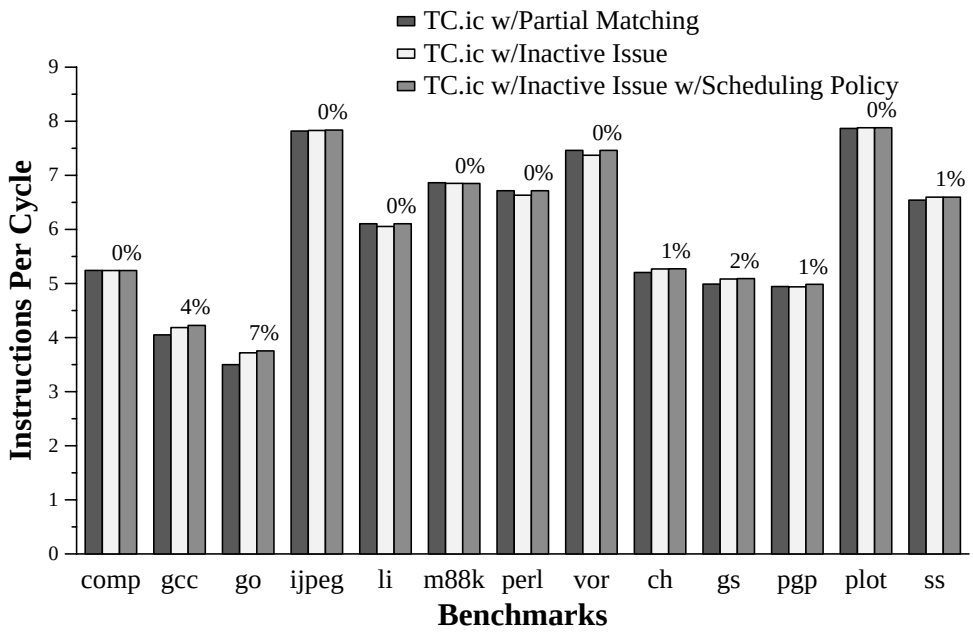

Figure 6.16: The performance of Inactive Issue with a new scheduling policy.

6.5 Dual Path Trace Segments

The trace caches explored in [37, 41, 43] contain trace segments which are dynamic sequences of blocks. Each subsequent block in a segment is a target of the previous block. Partial Matching enables another option: a trace segment need not consist of blocks from a single path of execution. The segment can be a multipath tree. The multiple branch predictor selects which path to issue from the fetched segment. If Inactive Issue is implemented, then the blocks on the non-selected paths are issued inactively.

In its simplest form, the multipath trace segment contains two separate paths, where the paths do not merge. Two cases of these simple dual path trace segments (along with normal single path segment) are shown in figure 6.17. The type-A segment is an example of a trace segment considered by the previous work on trace caches. The type-B and type-C segments are the new simple dual-path cases. Here, the selected path is issued actively and the other path is issued inactively. If the selected path was incorrect and the inactive path was correct, then the inactive instructions need not be refetched.

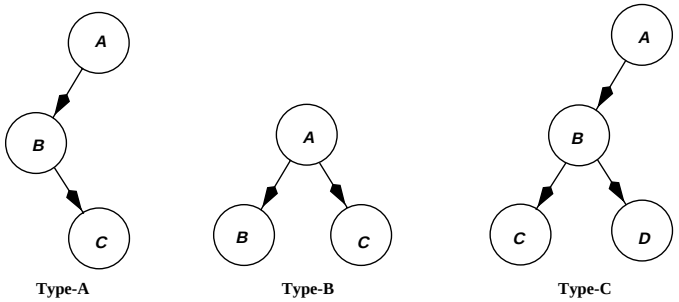

Figure 6.17: Three types of simple trace segments are shown here. Type-A contains only one path. Type-B and type-C trace segments encapsulate two paths.

6.5.1 Implementation issues

Type-A segments can be trivially constructed by the fill unit. To create type-B and type-C segments requires an enhancement to the fill algorithm: if an inactive (or discarded, if filling at retire) block fits into the pending segment, add it. Any subsequent inactive (discarded) blocks are flushed. The next active block will share the same predecessor block as the inactive (discarded) block. If the active block fits, add it and finalize. A simple dual path trace segment is created. If it doesn't fit, flush the inactive (discarded) block and add

the active block as normal.

Each trace cache line must now contain five target addresses, since the type-C segment has five possible successors. Furthermore, the fetch mechanism must be able to distinguish which instructions belong to which path thereby allowing the instructions along the predicted path to be actively issued. This requires that each instruction contain a two-bit field indicating the directions of the previous branches in the segment. This field is compared with the prediction to decide whether the instruction should be actively or inactively issued.

6.5.2 Measurement

The measurements done for this enhancement were performed using an older version of the simulator. The older simulator was based on the SimpleScalar ISA, as opposed to the Alpha AXP ISA supported by the newer simulator. While the absolute performance in IPC between the two simulator environments are likely to be different, the relative findings presented here are expected to be similar for both environments. Both the Inactive and Dual Path configurations were simulated on the SimpleScalar ISA.

The performance in percent speedup in IPC over Inactive Issue of a fetch mechanism which allows type-A, type-B, and type-C segments is shown in Table 6.4. Only the TC.ic configuration was measured. The performance of compress, go, gcc, and ijpeg is slightly boosted over the Inactive Issue case. While some benchmarks benefit from the Dual Path segments, others do not. The benchmarks li and m88ksim suffer a degradation in performance. If branches within a segment are strongly biased, then it is better to have more blocks on the likely path than to have blocks from each path. Forks within a segment are most useful when the corresponding branch oscillates, or is hard-to-predict.

6.5.3 Analysis

The Type-B segment is more commonly created than the Type-C segment. However, creating a Type-B segment potentially results in fewer instruction being fetched if the internal branch is biased towards one direction. If it is biased, then having both paths does not offer benefit; in this situation, it would be more advantageous to have a regular Type-A segment along the more likely path.

To make effective use of dual path segments, the fill unit must decide when a path should contain a fork and when it should not. Forks should appear when the adjoining

Benchmark	Dual Path Speedup over Inactive Issue
compress	1.0%
gcc	1.3%
go	1.0%
ijpeg	1.2%
li	-0.9%
m88ksim	-2.8%
perl	0.0%
vortex	0.0%

Table 6.4: The percent speedup in IPC of using Dual Path segments over Inactive Issue alone, measured on the TC.ic configuration using the SimpleScalar ISA.

branch oscillates over a very short span or if the branch is hard to predict. Hardware schemes, such as a confidence mechanism proposed by Jacobsen [23], can assist the fill unit by identifying branches which the branch predictor is not predicting well. Whenever such a branch is encountered, the pending segment is a possible candidate for a Dual Path.

6.6 Branch Promotion

The central benefit of the trace cache is its ability to boost the effective fetch rate beyond a single fetch block. With Partial Matching and Inactive Issue, the trace cache delivers approximately 12 instructions per cycle, or about two fetch blocks. Figure 6.18 is a histogram in which instruction fetches on the correct execution path are categorized by size. The data for this histogram was collected on the TC.ic configuration implementing Partial Matching and Inactive Issue and running the benchmark gcc. The conditions which limit the size of each fetch and their frequencies are identified on the graph.

There are seven conditions which can limit a fetch:

1. Partial Match. The path predicted by the branch predictor differed from the path of the trace segment and only a portion of the segment was subsequently issued.

2. Atomic Block. The fill unit was forced to create a segment smaller than maximum size because the subsequent block in the retire stream was larger than the space remaining in the pending segment. Here the fill unit is treating fetch blocks atomically.

3. ICache. The fetch was serviced by the icache and it was terminated by a control instruction or a cache line boundary [1] before 16 instructions were fetched.

4. Mispred BR. A mispredicted branch terminated the fetch. All inactively issued instructions within the segment after the branch also contribute to the size of the current fetch.

5. Max Size. The trace segment or icache fetch contained 16 instructions.

6. Ret, Indir, Trap. Returns, indirect jumps, and traps cause the pending segment to be finalized.

7. Maximum BRs. The fill unit created a segment containing three branches and all three were on path and issued actively from the current fetch.

Figure 6.18 shows that a large number of fetches are limited in size by the maximum branch limit of 3.

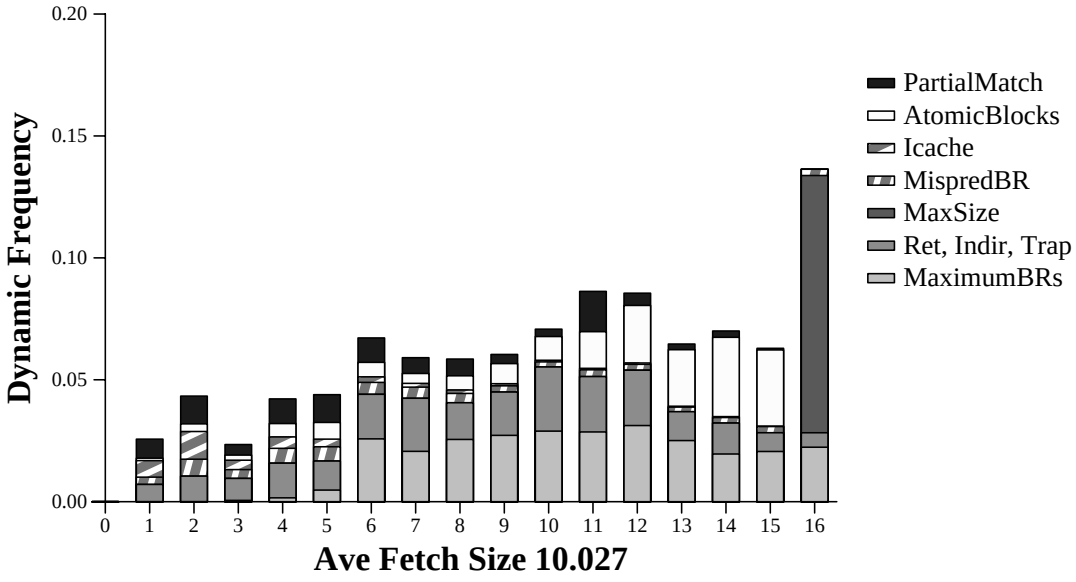

Figure 6.18: The fetch width breakdown for gcc on the TC.ic configuration.

In order to address this limitation in an effective manner, a frequently reported characteristic of conditional branches is drawn upon: during execution, over 50% of conditional branches are strongly biased [6]. When such a branch is detected, the branch will be converted by the fill unit into a branch with a built-in static prediction. This process is called

[1] The instruction cache implements split-line fetching, however cache line boundaries can still terminate a fetch if the request for the second line misses in the cache.

Branch Promotion. The concept is similar to branch filtering proposed by Chang et al [5]. A promoted branch requires no dynamic prediction and therefore need not consume branch predictor bandwidth when it is fetched. Its likely target instruction is either included within the trace segment or will be fetched in the subsequent fetch cycle. Two types of promoted conditional branches can be dynamically created—ones that are strongly biased towards not taken and ones strongly biased towards taken. With two bits, the fill unit can encode a branch as promoted and designate its likely outcome.

6.6.1 Implementation issues

Candidate branches can be detected via a hardware mechanism similar to the mechanism used for branch filtering. A table, indexed by branch address, called the branch bias table, is shown in Figure 6.19. It contains the previous outcome of the branch and the number of consecutive times the branch has had that same outcome. The bias table is updated whenever a branch is retired. The fill unit indexes into the bias table whenever a conditional branch is added to the pending segment. If the number of consecutive outcomes of the branch is beyond a *threshold*, the branch is promoted. For the experiments involving promoted branches presented here, the size of the bias table was fixed at 8K entries. A tagged bias table is modeled.

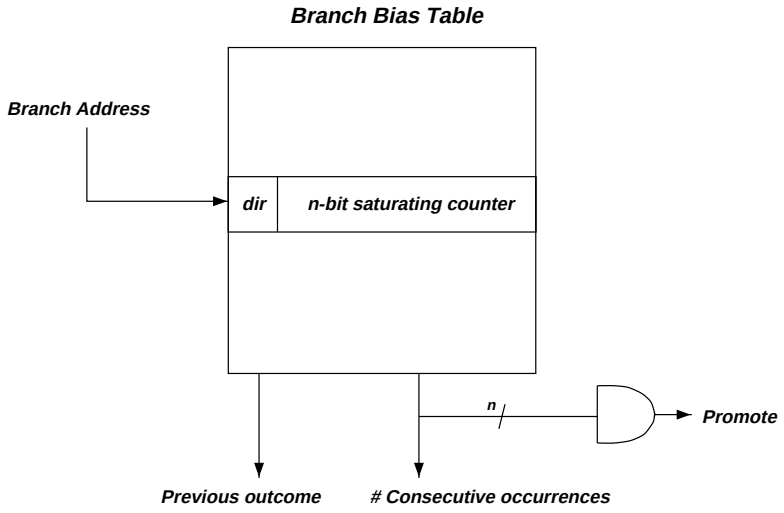

Figure 6.19: Diagram of the branch bias table.

A promoted branch that is mispredicted is said to *fault*, meaning that the block containing the branch must be undone. Processor state is reverted back to the previous branch, and execution proceeds from there with the promoted branch following the correct direction.

A faulting promoted branch is only demoted back to a normal conditional branch if there are two consecutive outcomes in the other direction or if there is a miss in the bias table [2]. The rationale for requiring two opposite outcomes is to inhibit the final iteration of a loop branch from demoting an otherwise strongly biased branch.

The baseline multiple branch predictor is not well utilized if branches are promoted. Most of trace segments will use only the first counter, leaving the remaining counters unutilized. To adjust for this, the pattern history table is restructured into three separated tables, each producing a single prediction. The first table contains 64K 2-bit counters and provides the prediction for the first branch. The second table contains 16K 2-bit counters and provides the prediction for the second branch. The third table contains 8K 2-bit counters and provides the prediction for the third branch. The total size of this predictor is 22KB.

6.6.2 Measurement

Figures 6.20– 6.22 show the performance measurements on the three trace cache configurations. Each figures shows the baseline configuration, the baseline configuration with Inactive Issue (and therefore Partial Matching), and a configuration with Inactive Issue and Branch Promotion. The threshold for Promotion for these measurements was set to 64. For the TC.ic configuration, the average IPC for the baseline is 4.79. For the Inactive Issue configuration, the average performance is 5.67 IPC. With Branch Promotion added, the performance goes to 5.76 IPC.

[2] Branch demotion actually occurs when two *or more* consecutive opposite outcomes are detected on a faulting promoted branch. This is because multiple copies of a branch can exist in the trace cache, and promoted copies are demoted only after they are fetched

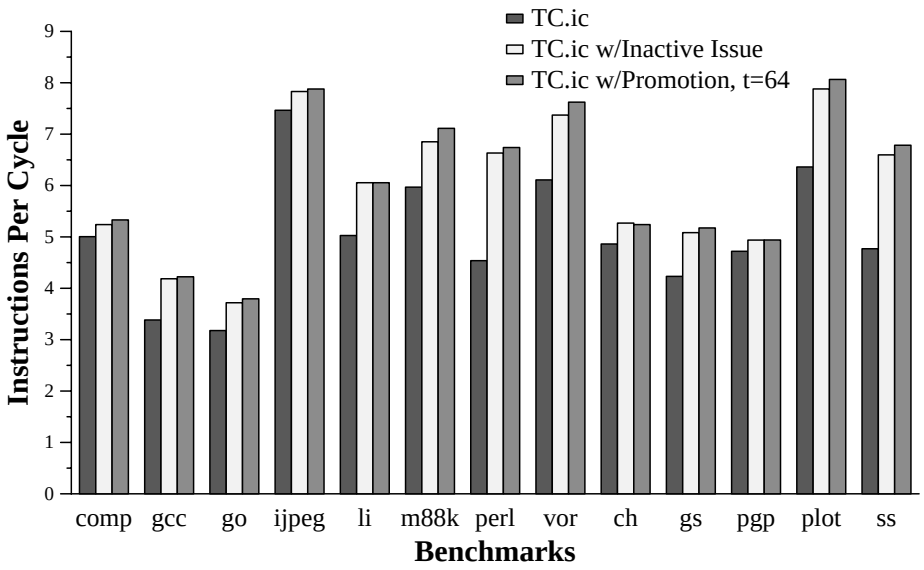

Figure 6.20: Performance of Branch Promotion on the TC.ic configuration. The configuration with Inactive Issue is shown for contrast.

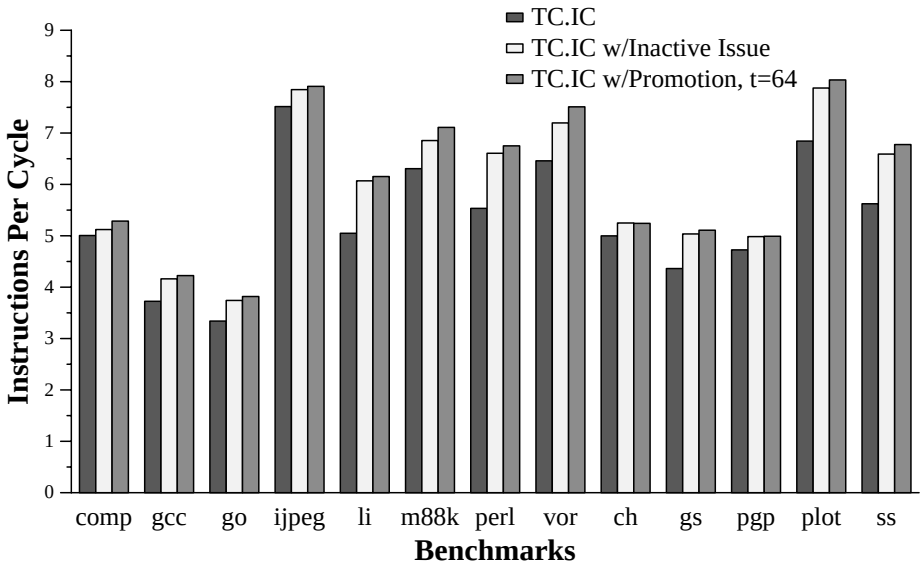

Figure 6.21: Performance of Branch Promotion on the TC.IC configuration. The configuration with Inactive Issue is shown for contrast.

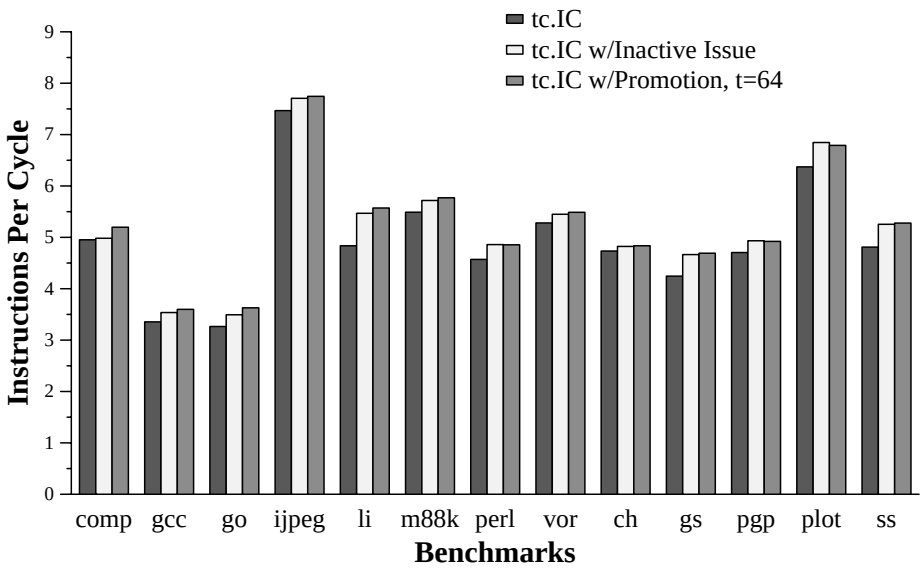

Figure 6.22: Performance of Branch Promotion on the tc.IC configuration. The configuration with Inactive Issue is shown for contrast.

6.6.3 Analysis

The effect on fetch rate

Table 6.5 shows the average effective fetch rate for Branch Promotion with various values for the *threshold* using the TC.ic configuration. Included are the effective fetch rates delivered with the Inactive Issue configuration described in section 6.4 as well as the two baseline icache configurations. The promoted branch configurations add Branch Promotion on top of Inactive Issue. For a threshold value of 64, the effective fetch rate is increased by a very slight average of 4% over the Inactive Issue configuration.

Configuration	Ave effective fetch rate
Single	6.50
Sequential	9.74
Basic TC.ic	9.65
Partial Matching	11.20
Inactive Issue	11.35
threshold = 8	11.70
threshold = 16	11.72
threshold = 32	11.72
threshold = 64	11.73
threshold = 128	11.70
threshold = 256	11.67

Table 6.5: The average effective fetch rate with Promotion. All trace cache configurations are based on the TC.ic model.

With Branch Promotion, many fewer fetches are limited by branch predictor bandwidth. Figure 6.23 is similar to figure 6.18 and shows a histogram of fetch sizes on the benchmark gcc annotated with reasons for fetch termination. The threshold for Branch Promotion is 64 consecutive occurrences. Compared to figure 6.18, there are fewer fetches terminated because of the maximum branch limit. For this benchmark, the effective fetch rate is 10.56 instructions per cycle, a 5% increase over using Inactive Issue alone.

Notice, however, that fetch size for gcc is being limited now by Atomic Blocks. Trace Packing, introduced in the next section, deals with this limitation.

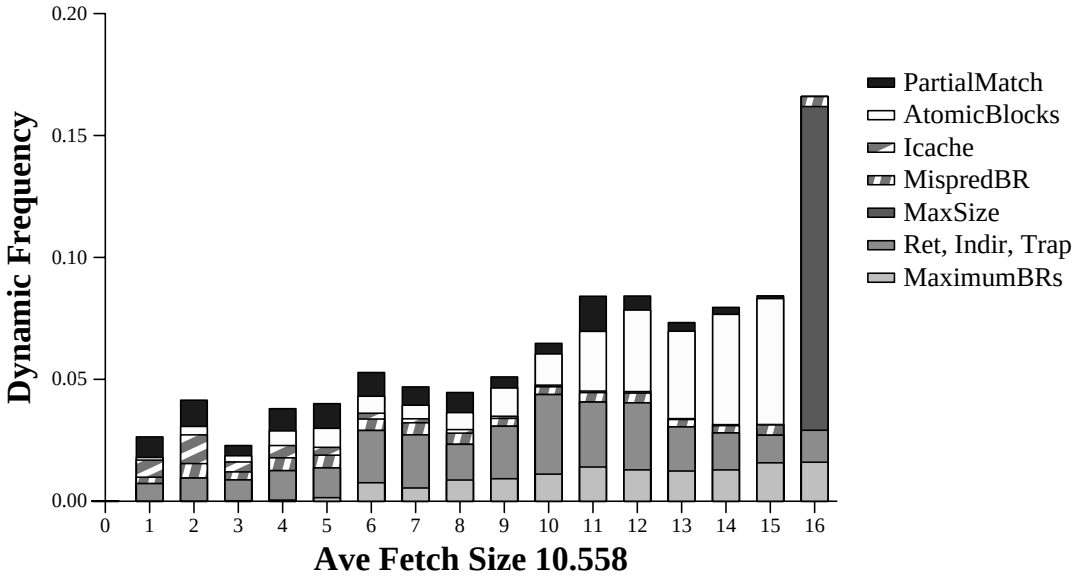

Figure 6.23: The fetch width breakdown for gcc on the TC.ic configuration with Branch Promotion.

The effect on branch prediction

Figure 6.24 displays the dynamic frequency of promoted branches, regular branches, and promoted branches which fault (threshold = 64). A significant number (about 60%) of dynamic branches are converted into promoted branches. Of these, a very insignificant number ever fault.

Branch Promotion removes a large number of easily predictable branches from the domain of the dynamic predictor. Since these branches do not update the predictor's pattern history table, interference [56] is reduced in a manner similar to branch filtering. Their outcomes, however, are added to the global branch history to maintain the integrity of the predictor's information. Because interference is reduced, prediction accuracy improves overall.

However, the benchmark executables have been optimized to reduce the incidence of taken branches, as mentioned in section 4.3. One beneficial side effect of this optimization is that global branch interference is reduced via an "agree" effect [49]. Since branch outcomes are likely to be not-taken, a certain amount of negative interference in the branch predictor's Pattern History Table is converted to positive interference. This happens because branches which collide in the PHT are more likely to force the counter into a not-taken state. If this optimization were not used to generate the executables, then the interference-reducing

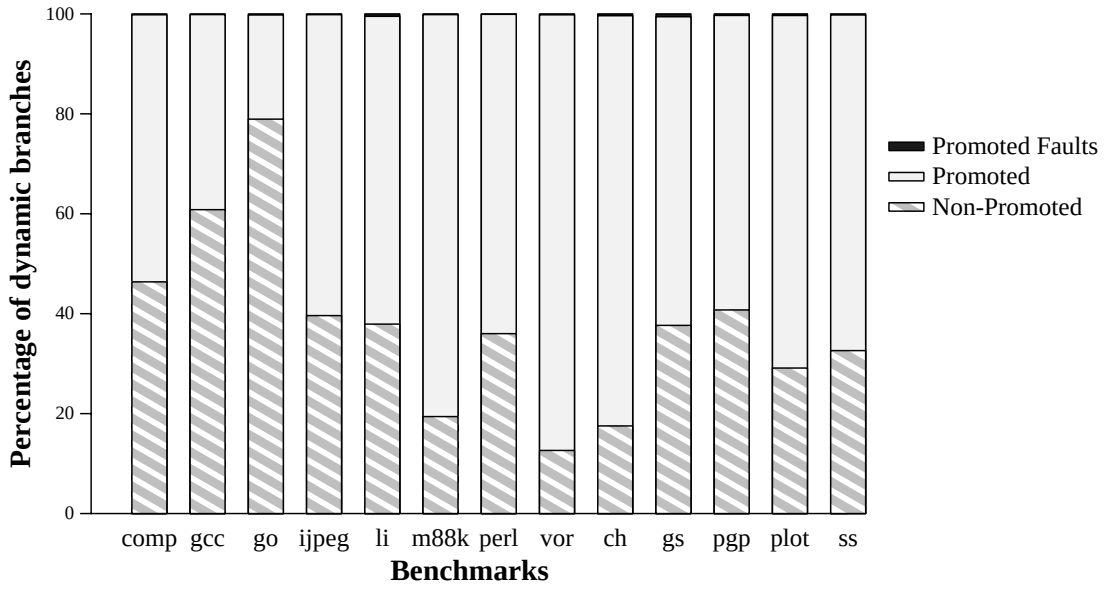

Figure 6.24: The frequency of promoted branches and how often they fault.

effect of Branch Promotion would be more pronounced.

Figure 6.25 shows the percent change in the number of conditional branches on the correct execution path which are mispredicted for three configurations (threshold 32, 64, and 128) compared to using Inactive Issue. In most configurations, the number of mispredictions is reduced, in some cases significantly. For go, Promotion at threshold=64 reduces the number of mispredicted conditional branches to about 92% of the Inactive Issue case. Overall, the branch misprediction rate drops from 6% on the configuration with inactive issue to 5% for threshold=64. However, premature Promotion can lead to frequent faulting (faults also count as mispredictions), as is the case with the benchmarks li, pgp, and plot. Increasing the threshold for Promotion reduces the effect of premature Promotion, as branches that pass the larger threshold are more likely to remain biased.

Creating Large Execution Units

Branch Promotion effectively enlarges the execution atomic unit (EAU) from the perspective of the trace cache. An EAU is a unit of instructions in which all instructions execute or none execute. Conditional branches terminate an execution atomic unit whereas promoted branches need not. If a promoted branch faults, the machine is backed up to the previous checkpoint (the end of the previous block) and then resumes with the promoted branch executing in the correct direction. The implications and measurement of EAUs will

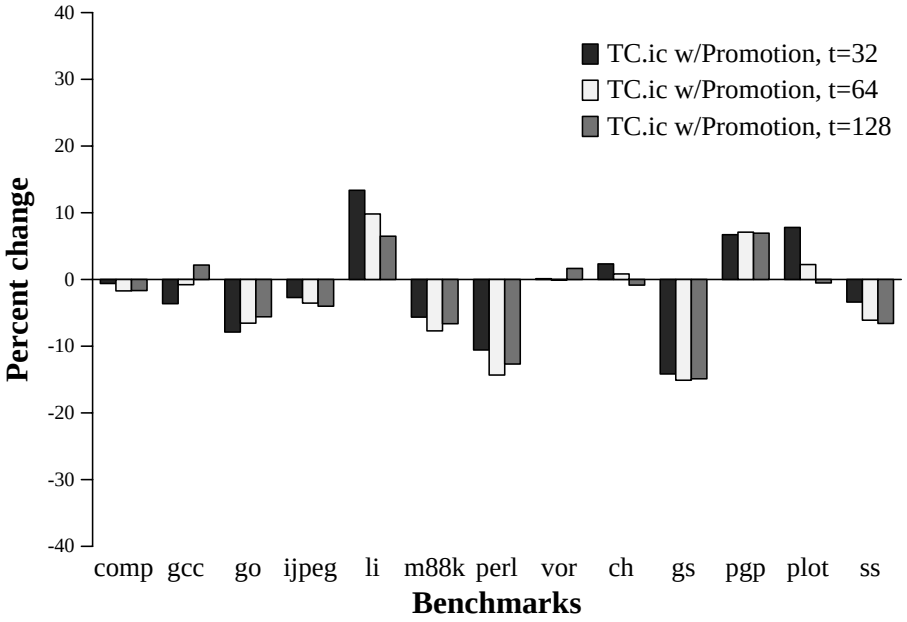

Figure 6.25: The percent change, relative to the Inactive Issue case, in the number of mispredicted branches when branches are promoted.

be covered in Section 8.4

Another result of Promotion is that fewer non-promoted branches are encountered per fetch of 16 instructions. Previous trace cache studies measured that at least three branches were required to effectively deliver bandwidth for a 16-wide machine. The data in Table 6.6 shows that with Promotion on average 91% of all fetches require only one dynamic branch prediction.

Configuration	0 predictions	1 predictions	2 predictions	3 predictions
Inactive Issue	30%	32%	21%	17%
threshold = 64	63%	28%	7%	2%

Table 6.6: The number of predictions required each fetch cycle, averaged over all benchmarks.

Branch Promotion can be done statically, as well. The ISA must allow for extra encodings to communicate strongly biased branches to the hardware. However, branches which switch outcomes during execution but remain biased or are sensitive to input data may be missed during static analysis. There are a few advantages. Branches need not go through

a warm-up phase before being detected as promotable and branches which have irregular behavior but are strongly biased can be more easily detected statically.

6.6.4 Extensions

The concept of Branch Promotion can also be applied to returns and indirect jumps. Here, the bias hardware monitors these instructions to determine if their outcomes are fixed to a single target address. When a particular indirect jump is detected as going to the same target a *threshold* number of times, it is promoted into static jump to that target. The semantics of the new promoted instruction ensure that if the target is different, then a fault will direct execution to the correct target. Table 6.7 provides evidence that such Promotion (return promotion, in particular) may be worthwhile. The table shows the percentage of all dynamic return and indirect jumps which have gone to the same target for 16 or more consecutive outcomes. Using this classification, almost 60% of all return instructions are pinned to the same target. Using Promotion for these cases will increase trace segment length, thus effective fetch rate, and reduce the pressure on the return address stack.

Benchmark	RETURN	INDIR JMP
compress	83.84%	43.56%
gcc	41.71%	12.43%
go	45.66%	7.09%
ijpeg	9.20%	96.45%
li	53.89%	7.38%
m88ksim	74.32%	0.67%
perl	54.84%	6.89%
vortex	47.82%	19.21%
chess	76.15%	2.73%
gs	80.58%	50.87%
pgp	82.91%	38.16%
plot	35.62%	46.24%
ss	55.54%	65.32%
Average	57.08%	30.54%

Table 6.7: The percentage of RETURNs and INDIR JMPs which have the same target for 16 or more consecutive previous occurrences.

6.7 Trace Packing

Branch Promotion results in a 5% percent increase in overall effective fetch rate over Inactive Issue. However from the fetch termination histogram in figure 6.23, it is evident that by promoting branches, fetch bandwidth is slightly increased, and quickly limited by the fact that fetch blocks are atomic entities within the fill unit. If fetch blocks are atomic, the fill unit will not divide a block of newly retired instructions across trace segments (unless the block is larger than 16 instructions). If the pending trace segment contains 13 instructions, then a block of 9 instructions will cause the segment of 13 instructions to be written and the block of 9 will begin a new segment.

There is a strong rationale for treating fetch blocks as atomic entities. Figure 6.26 shows a loop composed of three fetch blocks. If blocks are treated atomically, three trace segments containing the loop blocks are formed in the steady state: AB, CA, BC. But if the fill unit is allowed to fragment a block, a process called *Trace Packing*, then six segments could potentially be created. [3] The problem gets significantly worse if there are different control paths within the loop.

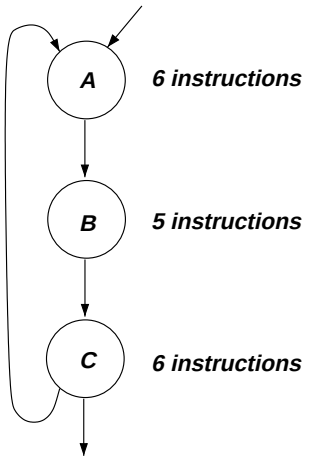

Figure 6.26: A loop composed of 3 fetch blocks.

On the positive side, this duplication potentially increases the delivered fetch rate, e.g., loops will be dynamically unrolled so that a maximum number of blocks can be fetched per cycle. But the primary cost of this duplication is increased contention for trace cache lines. With non-atomic treatment of blocks, trace segments can be packed with more instructions,

[3] The six segments are $A_6B_5C_5$, $C_1A_6B_5$, $C_6A_6B_4$, $B_1C_6A_6$, $B_5C_6A_5$, $A_1B_5C_6$. The subscripts denote the number of instructions of each block included in the segment. Notice that even with trace packing, no instructions beyond the third branch are added to the pending segment

but at the cost of increased duplication within the trace cache.

6.7.1 Measurement

Figures 6.27–6.29 show the performance of Trace Packing. In all experiments performed, Trace Packing was added on top of Inactive Issue. Overall, Trace Packing yields a 2% performance improvement over Inactive Issue on the TC.ic configuration.

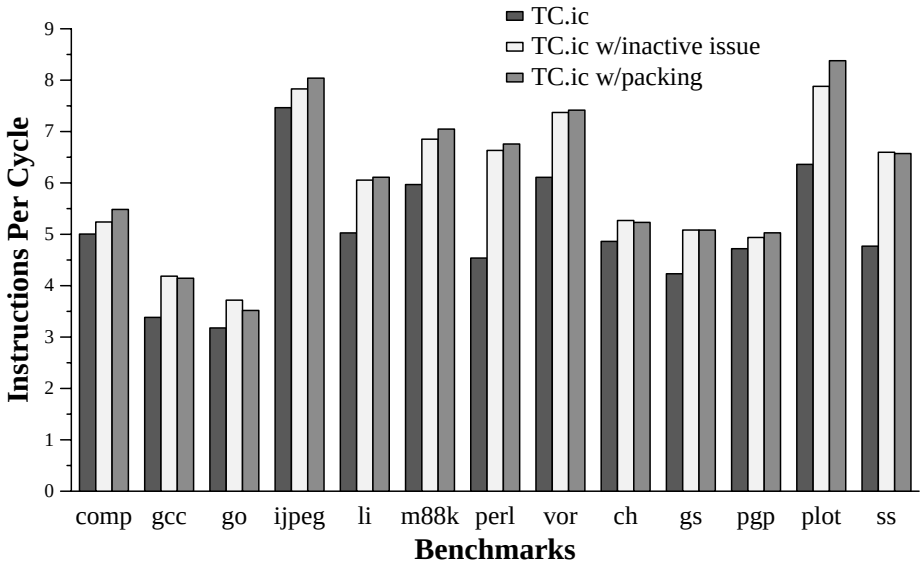

Figure 6.27: Performance of Trace Packing on the TC.ic configuration. The configuration with Inactive Issue is shown for contrast.

6.7.2 Analysis

Effect on fetch rate

Both Branch Promotion and Trace Packing are limited by the problem the other is solving. The key to unlocking the fetch potential of the trace cache is to use both techniques. Figure 6.30 compares the effective fetch rate for four TC.ic configurations: with Inactive Issue, with Inactive Issue and Branch Promotion, with Inactive Issue and Trace Packing, and with all three. The threshold for Branch Promotion is set to 64. With both Branch Promotion and Trace Packing in place, the fetch bandwidth is boosted by an average of 17% across all the benchmarks simulated over Inactive Issue alone. The percentages above the bars indicate the increase with both Packing and Promotion over the Inactive Issue case. In many cases, the total increase is more than the sum of the individual increases, most notably

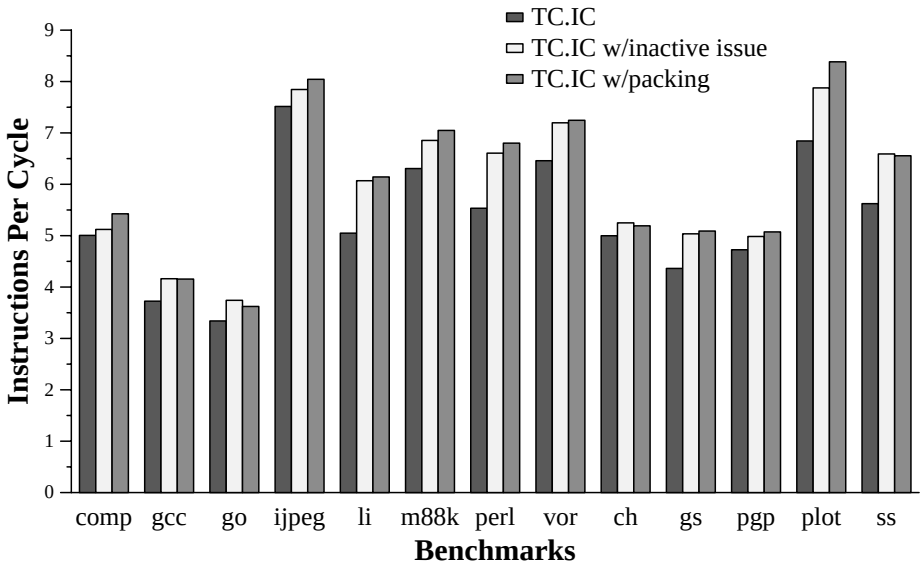

Figure 6.28: Performance of Trace Packing on the TC.IC configuration. The configuration with Inactive Issue is shown for contrast.

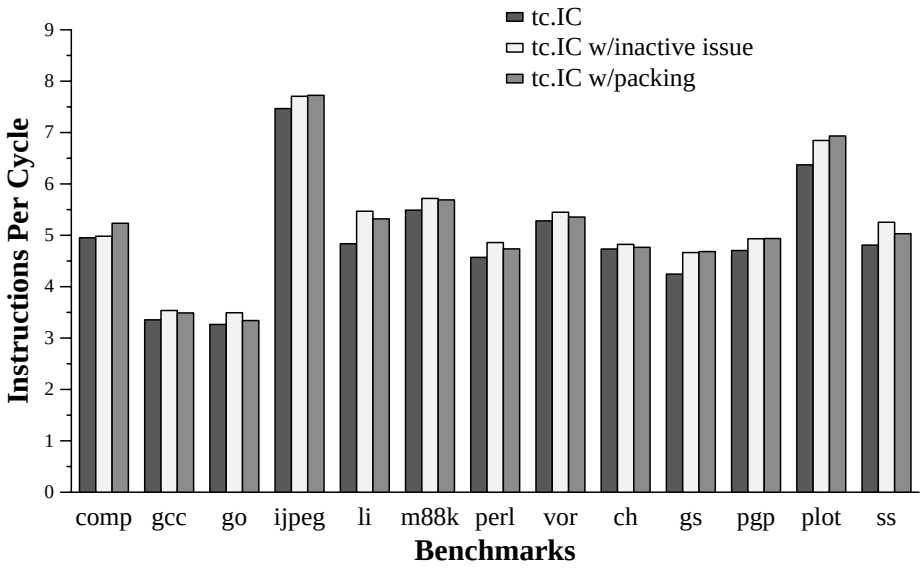

Figure 6.29: Performance of Trace Packing on the tc.IC configuration. The configuration with Inactive Issue is shown for contrast.

with the benchmarks li, perl, chess, pgp, and, simplescalar, and on the overall average. With the benchmark go, the average fetch size only increases by 4%. Since this benchmark suffers from both a high number of trace cache misses and poor branch prediction, many fetches are limited by the delivery rate of the icache and by mispredicted branches.

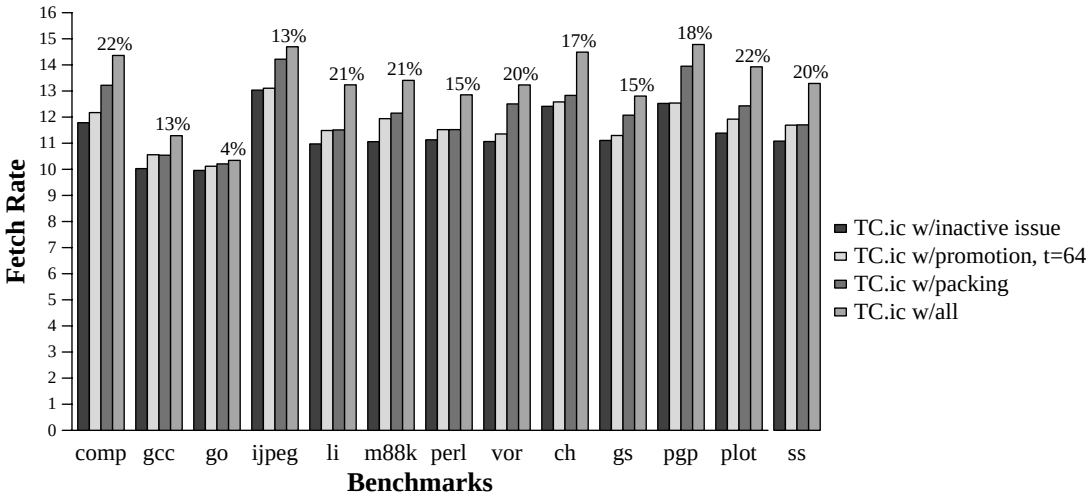

Figure 6.30: The effective fetch rates for all techniques.

Effect on cache miss rate

Due to a large increase in duplication, the downside of Trace Packing is that it increases the number of fetch requests which miss in the trace cache. Table 6.8 contrasts the trace cache misses per 1000 instructions for the Inactive Issue scheme with that of Trace Packing. Both were measured on the TC.ic configuration. With Trace Packing, the miss rate, on average, increases by 76%. The benchmarks vortex and ghostscript, which suffer a significant number of misses to begin with, more than double their miss rate with Trace Packing. Some benchmarks (ijpeg, chess, ss) suffer fourfold to fivefold more misses with Trace Packing. This increase in cache miss rate is the primary reason why some benchmarks suffer a loss of performance with Trace Packing (gcc, go, ch, gs, ss. See Figure 6.27)

As will be demonstrated in Section 8.3, Trace Packing increases the number of copies of an instruction which are resident in the trace cache concurrently. The measurements indicate that Trace Packing increases the average number of copies per instruction from about three to about eleven.

6.7.3 Extensions

As mentioned, a serious drawback to Trace Packing is the effect block fragmentation has on trace cache contention. The increased cache contention problem stems from the fact that trace segments now can begin at any instruction. In configurations where blocks are treated atomically, trace segments are naturally *synchronized* at fetch block boundaries. Either an

Benchmark	Inactive Issue	Trace Packing
compress	0.01	0.01
gcc	6.00	9.21
go	14.59	22.75
ijpeg	0.01	0.06
li	0.02	0.04
m88ksim	0.13	0.32
perl	0.26	0.67
vortex	1.15	3.31
chess	0.18	0.88
gs	1.49	3.35
pgp	0.04	0.06
plot	0.41	0.36
ss	0.35	1.44
Average	1.41	2.48

Table 6.8: The trace cache miss rates in misses per 1000 instructions for the TC.ic configuration with Inactive Issue and with Trace Packing.

entire block is replicated or nothing from that block is replicated. With Trace Packing, trace segments are no longer synchronized at fetch block boundaries and the instruction duplication grows significantly. To deal with this problem, two schemes where the fill unit only packs n instructions at a time are examined. For example, if n=2 and the entire block doesn't fit, then only an even number of instructions are added to the pending segment. Blocks now fragment at half the number places as compared to unregulated trace packing.

Another scheme only packs traces if the pending trace segments are less than (or equal to) half full OR the pending segment contains a backwards branch with displacement of 32 or fewer instructions. First, Trace Packing is most important when the number of unused instruction slots in a trace segment is large. Not only will valuable space on the cache line go unused when such a segment is stored in the cache, but whenever this segment is fetched, fetch bandwidth will be wasted. By only packing a trace when the amount of unused space is high, fragmentation costs are incurred only when the potential payoff is high. Second, the potential payoff is higher for small tight loops. Since dynamic unrolling may significantly boost fetch bandwidth for a small loop, the benefit may overcome the costs of fragmentation, particularly if the number of iterations is high. This scheme is referred to as *cost-regulated* Trace Packing.

The percentage change in cache misses for unregulated Trace Packing, regulating at every other instruction (n=2) and every fourth instruction (n=4) and the cost-regulated scheme is provided in Table 6.9 on the TC.ic configuration. Also listed in this table is the average effective fetch rate attained for each form of Trace Packing. The cost-regulated scheme provides a good middle ground, providing high fetch rates with low miss rates.

Benchmark	Trace Packing	$n = 2$	$n = 4$	cost-reg
compress	0.01	0.01	0.01	0.01
gcc	9.21	6.88	6.72	7.04
go	22.75	16.52	16.34	16.60
ijpeg	0.06	0.01	0.01	0.01
li	0.04	0.02	0.03	0.02
m88ksim	0.32	0.08	0.19	0.10
perl	0.67	0.58	0.38	0.65
vortex	3.31	1.58	1.65	1.81
chess	0.88	0.25	0.25	0.35
gs	3.35	1.92	1.85	2.04
pgp	0.06	0.04	0.04	0.08
plot	0.36	0.13	0.15	0.10
ss	1.44	0.64	0.65	0.67
Average	2.48	1.63	1.62	1.73
Ave Eff Fetch Rate	12.56	12.16	12.07	12.22

Table 6.9: The trace cache miss rates in misses per 1000 instructions for various flavors of Trace Packing.

The techniques for regulating instruction duplication within the trace cache are crucial if the size of the fetch mechanism is smaller than 128KB. The experiments presented in this section are done on a fetch mechanism of 128KB and simulated on benchmarks which may not be representative, in terms of size, to commercial applications. The lost fetch bandwidth due to cache misses is fairly minor. If the fetch mechanism is smaller, or if the latency to the second level instruction cache is higher, or if the applications have larger icache footprints, such techniques to regulate duplication may be necessary for higher performance. In Section 8.3, precise measurements of instruction duplication within the trace cache are provided, along with techniques for limiting it by exploiting good synchronization points at which to terminate trace segment expansion.

6.8 All Enhancements Combined

In this section, the four fetch rate enhancements which impacted bottom-line performance—Partial Matching, Inactive Issue, Branch Promotion, and Trace Packing—are combined and overall performance is measured and analyzed. These four enhancements represent an aggressive implementation of the trace cache mechanism. The objective of this section is to demonstrate the performance of this implementation in relation to the instruction cache baselines.

6.8.1 Measurement

Figures 6.31 and 6.32 show the measured performance of the five configurations, divided in two graphs. The first graph shows the SPECint95 benchmarks and the second the UNIX applications. These two figures are similar to Figures 5.1 and 5.2 from the previous chapter—the basic difference is that the trace cache configurations measured here are enhanced with the techniques presented in this chapter. Without the techniques, as reported in Section 5.1, the Sequential-Block ICache performs best by 2% over the best trace cache configuration. With the techniques, however, the trace cache configuration TC.ic performs consistently best and outperforms the Sequential-Block ICache by 14% and the Single-Block ICache by 33%.

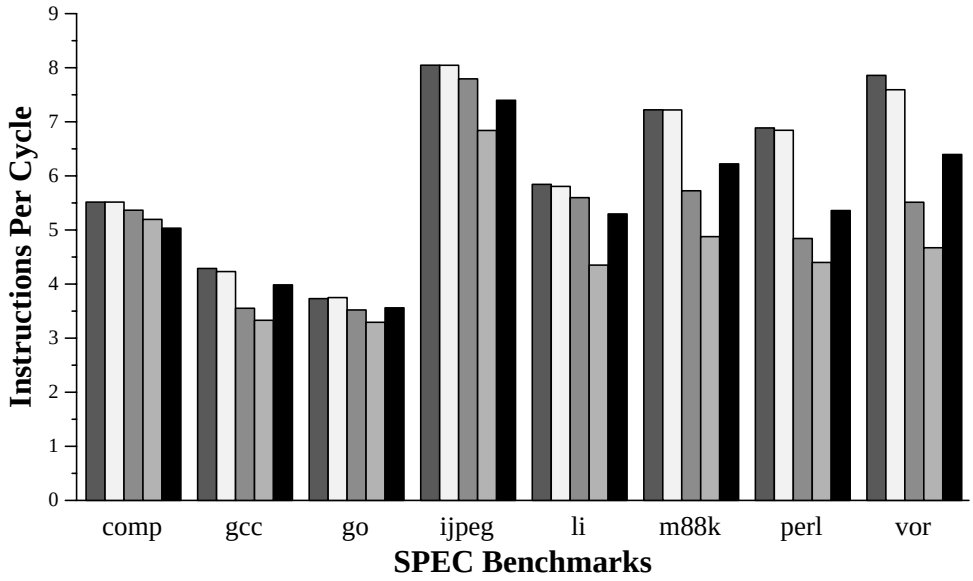

Figure 6.31: The performance in IPC of the five configurations on SPECint95.

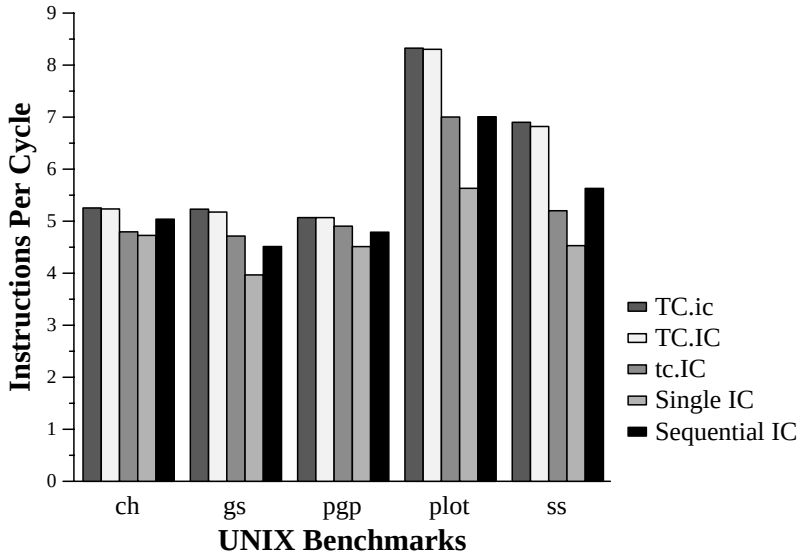

Figure 6.32: Performance of the five configurations on the UNIX benchmarks.

6.8.2 Analysis

An analysis of the new, enhanced mechanism is provided in this section to help identify positive and negative factors on performance. The analysis is focused around the three factors which affect fetch engine performance: fetch rate, branch misses, and cache misses. In Chapter 8 global analysis of the trace cache mechanism is presented.

Effect on fetch rate

As demonstrated in the various sections of this chapter, the main benefit of the trace cache enhancements is that they boost effective fetch rate. These enhancements allow the trace cache to service more requests (e.g., Partial Matching), and to deliver more instructions for each request it does service (e.g., Trace Packing). Figure 6.33 shows the increase in effective fetch rate with the Enhanced TC.ic configuration versus the baseline TC.ic configuration and versus the Sequential-Block ICache. The percentage difference between the Enhanced TC.ic and ICache configurations are displayed on top of the bars. Overall, the enhanced trace cache attains a 34% increase in fetch rate over the ICache and a 35% boost over the baseline TC.ic configuration.

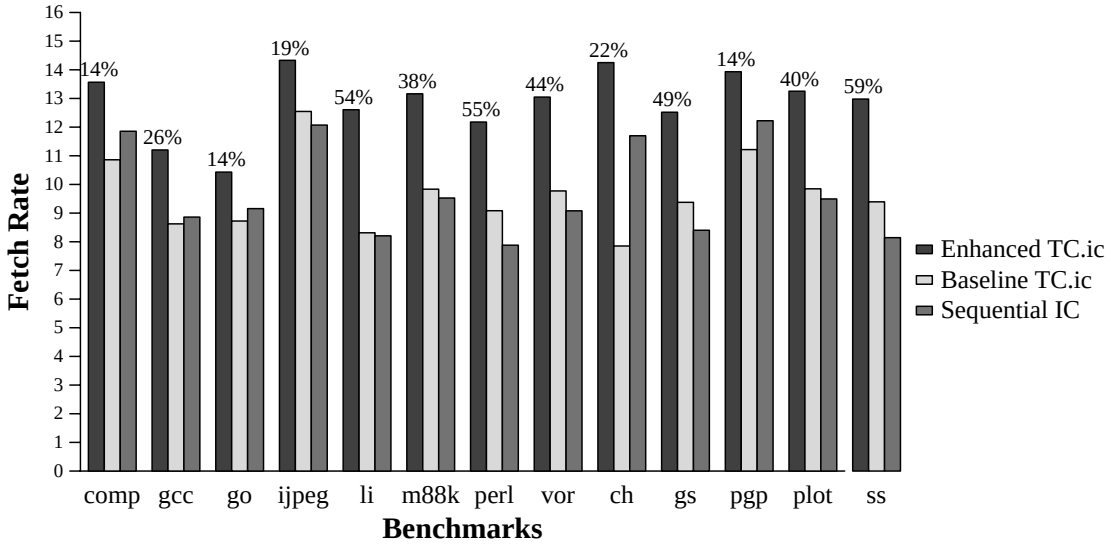

Figure 6.33: The effective fetch rate of the Enhanced TC.ic configuration versus the baseline TC.ic and Sequential-Block ICache.

Effect on cache miss rate

A secondary, but still significant, benefit provided by the combined enhancements is a reduction in trace cache misses. For example, the Partial Matching enhancement relaxed the trace cache hit policy increasing the number of requests serviced by the trace cache. Table 6.10 is similar to Table 5.5. The left-hand side of the table lists the trace cache misses per 1000 instructions, and the right-hand side lists both trace cache and icache misses combined. Comparing the averages of this table and Table 5.5, it is notable that

the trace cache miss rate drops by a factor of 18 for the TC.ic configuration. Furthermore for the same configuration, the number of overall fetch misses drops by a factor of 4. The rates are lower overall despite the duplication effect from the cost-regulated Trace Packing. Not only is the trace cache being used more effectively, but the small supporting instruction cache is better at fulfilling requests which are not in the larger cache. Overall, the trace cache configurations have larger fetch miss rates than the icache configurations but the rates are significantly low that the effect on bottom-line performance is small.

Benchmark	TC.ic	TC.IC	tc.IC	TC.ic	TC.IC	tc.IC	Single	Seqntl
compress	0.01	0.01	3.94	0.00	0.00	0.00	0.00	0.00
gcc	7.20	15.79	100.45	4.03	1.99	0.92	0.97	0.95
go	17.00	22.52	64.08	5.69	3.37	1.05	0.97	1.04
ijpeg	0.01	0.02	13.44	0.00	0.00	0.00	0.00	0.00
li	0.04	0.09	29.04	0.00	0.00	0.00	0.00	0.00
m88ksim	0.05	0.42	59.27	0.04	0.00	0.00	0.00	0.00
perl	0.43	2.36	102.13	0.30	0.05	0.04	0.03	0.04
vortex	1.73	6.27	88.59	1.27	1.04	0.42	0.45	0.43
chess	0.45	2.01	33.00	0.23	0.02	0.00	0.00	0.00
gs	2.25	5.63	58.27	1.38	1.33	0.27	0.52	0.51
pgp	0.04	0.06	8.67	0.01	0.01	0.00	0.00	0.01
plot	0.51	0.78	40.82	0.05	0.03	0.01	0.01	0.01
ss	0.71	2.66	81.66	0.42	0.05	0.02	0.01	0.02
Average	1.77	3.53	42.45	0.81	0.50	0.16	0.19	0.19

Table 6.10: The left half of this table lists the trace cache misses per 1000 instructions. The right half lists total fetch misses (tcache+icache) per 1000 instructions.

Effect on branch misses

Table 6.11 lists the conditional branch misprediction rates for the five test configurations, similar to Table 5.3 from the previous chapter. As with the baseline configurations of the previous chapter, the Single-Block Icache has a significantly lower branch misprediction rate due to the ability of the hybrid predictor to advantage of both per-address and global branch correlation. Compared with the baseline miss rates, the enhanced TC.ic configuration has a slightly lower misprediction rate primarily due to the interference reduction of Branch Promotion. Most benchmarks benefit from Promotion, whereas some

suffer from frequent faulting due to bad Promotions, such as the benchmarks li and plot. Some benchmarks also suffer from the negative effects of duplication brought on by Trace Packing, such as the benchmark gcc. The absence of duplication enables the Sequential-Block ICache to outperform the TC.ic configuration with respect to prediction accuracy. With the Sequential-Block ICache, each branch is coupled to fewer fetch groups, therefore each branch "touches" fewer counters in the pattern history table.

Benchmark	TC.ic	TC.IC	tc.IC	Single	Seqntl
compress	8.00	8.00	8.15	5.00	8.49
gcc	7.81	8.01	8.24	5.98	7.39
go	16.45	16.49	16.63	14.29	16.32
ijpeg	9.02	9.02	9.35	8.46	9.14
li	5.70	5.78	4.57	2.72	4.26
m88ksim	1.81	1.79	2.50	0.79	2.00
perl	2.34	2.36	2.67	1.24	2.76
vortex	1.43	1.53	2.43	0.74	1.34
chess	2.47	2.50	2.57	1.83	2.25
gs	5.16	5.25	6.07	4.16	5.44
pgp	5.04	5.04	5.83	4.35	5.27
plot	2.80	2.79	3.01	1.53	2.63
ss	4.33	4.40	5.23	3.36	4.61
Average	5.57	5.61	5.94	4.19	5.53

Table 6.11: The conditional branch misprediction rates (in percentage) of the five configurations.

Cycle breakdown

Finally, a cycle breakdown is presented for each of the benchmarks running on the enhanced TC.ic configuration. Figure 6.34 is similar to the Figures 5.3 and 5.4 from the previous chapter. Each fetch cycle is categorized into one of six categories viewed from the perspective of the fetch engine. See Section 5.2.

The basic observation from this diagram is that a number of benchmarks are affected by more full window stalls. Either the execution backend is unable to consume instructions at a fast enough rate due to structural bottlenecks OR the application has regions of limited parallelism where a larger window is required to buffer the incoming instructions.

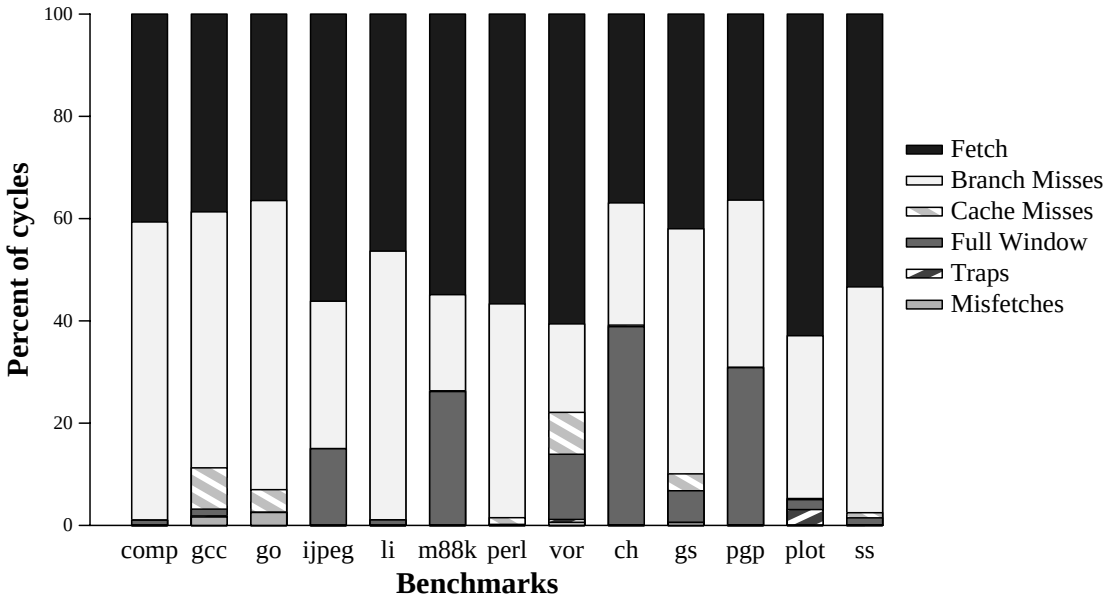

Figure 6.34: A fetch cycle breakdown for each of the benchmarks on the enhanced TC.ic configuration.

6.9 Summary

In this chapter, several trace cache enhancements were described, measured, and analyzed.

Partial Matching is a technique which relaxes the trace cache hit criterion allowing partial segments to be supplied in response to a request. Partial Matching was determined to be a significant impact on performance.

Path Associativity allows multiple trace segments starting at the same address, but containing different paths, to be stored concurrently in the trace cache. The branch prediction is used select which of potentially several trace cache lines to supply. Path Associativity was found to not affect performance significantly because of the limited number of trace segments which benefit from this enhancement.

Inactive Issue is a hedge against branch mispredictions. Inactive Issue allows the issue of instructions on a trace cache line but not on the path selected by the branch predictor. These instructions are issued "inactively". They execute but their results remain invisible until it is determined that they indeed should have been on the correct execution path. Inactive Issue is a optimization of Partial Matching and offers a performance advantage on benchmarks with poor branch prediction performance.

Branch Promotion is a technique which converts highly biased branches into unconditional branches with a faulting semantic. Promoted branches produce a hardware fault when they are detected to behave contrary to their promoted direction. Branch Promotion increases the number of instruction which can be put into a trace segment.

Trace Packing is the generic technique of placing instructions into a pending trace segment while ignoring fetch block boundaries. With Branch Promotion, Trace Packing significantly increase the trace cache's effective fetch rate.

Enhanced with the techniques of Partial Matching, Inactive Issue, Branch Promotion, and Trace Packing, the trace cache performance jumps by 22% over a simple trace cache. The enhanced TC.ic configuration attains a 14% performance increase over the Sequential-Block ICache.

CHAPTER 7

Sensitivity Studies

The objective of this chapter is to examine the sensitivity of the high performance mechanism developed in the previous chapter to changes in several basic design parameters of both the trace cache and of the other components of the fetch mechanism. A range of studies were performed, from the set-associativity of the trace cache, to the aggressiveness of the instruction cache pipeline, to the design and latency of the fill unit. Also included is an examination of the impact of simple variations in branch predictor design on overall performance. For all studies in this chapter, the trace cache configurations use Partial Matching, Inactive Issue, Branch Promotion, and cost-regulated Trace Packing, as described in the previous chapter.

7.1 Set-Associativity of the Trace Cache

For the following experiment, the set associativity of the enhanced TC.ic configuration is varied from 1 to 16. The results are presented in Figure 7.1. Even with the large capacity of the 128KB trace cache, there is a considerable performance boost from increasing set associativity. The percentages above the 4-way bars indicate the performance boost of a 4-way trace cache over a direct-mapped one.

This high reliance on set-associativity (in contrast to an equivalently large icache) is due to the manner in which the trace cache is indexed and is organized. With a 2K entry trace cache, two traces which are 2048 instructions apart will map to the same trace cache set. With a similarly sized instruction cache, instructions must be 32K instructions apart before they map to the same set. The mapping of the trace cache is more susceptible to thrashing

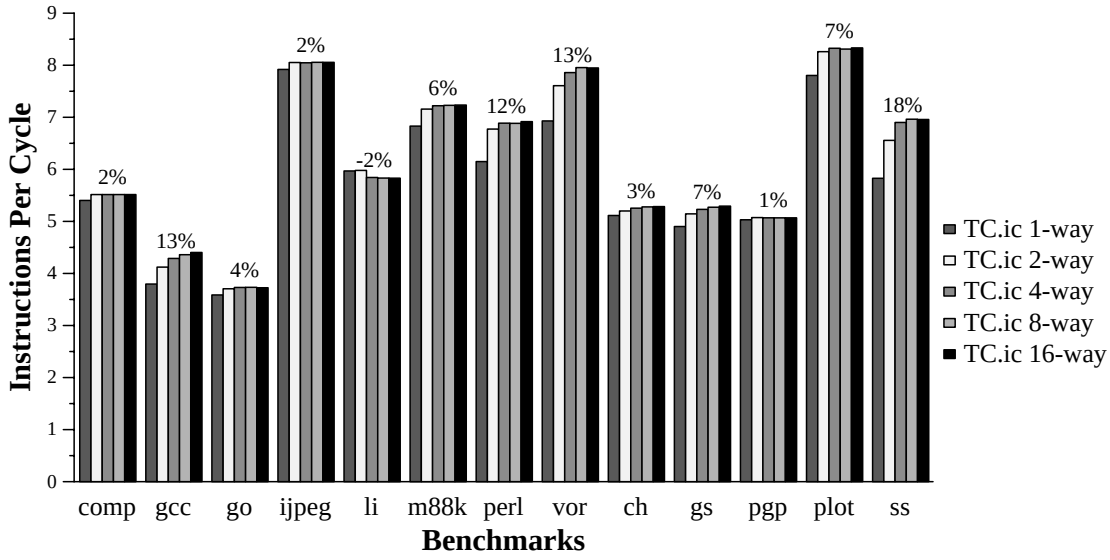

Figure 7.1: The TC.ic performance with varying degrees of trace cache associativity.

due to the natural spatial behavior of programs—branch targets are more likely to be nearby. A study performed by Smith [47] has confirmed this sensitivity to set-associativity in Branch Target Buffers, which are indexed similarly to trace caches.

The average trace cache miss rate per 1000 instructions (averaged over all benchmarks) is given in Table 7.1. Notice that trace cache associativity has a very strong effect on miss rate.

	direct	2-way	4-way	8-way	16-way
Ave Miss Rate	7.76	2.63	1.77	1.59	1.50

Table 7.1: The effect of set-associativity of trace cache miss rate.

Finally, several benchmarks exhibit a drop in performance as associativity is increased. The benchmark li, for example, drops in performance as the associativity of the trace cache is increased beyond 2-way. Two factors are at work here. First, the benchmark li suffers from a significant amount of duplication in the trace cache. Second, the benchmark suffers from a higher rate of faulting of promoted branches. The duplication causes several copies of a promoted branch to be cached concurrently. If a copy of a promoted branch is fetched and subsequently faults and is demoted, it will be written back as a regular conditional branch. However, the other copies of that branch, some of which are likely to be promoted, remain unchanged. As a result, if they are fetched, they are likely to fault. An increase in

set-associativity increases the likelihood that several copies of these branches will remain resident. At lower associativities, promoted branches are more likely to get displaced, and thus be treated as normal conditional branches.

7.2 Trace Cache Write Policy

When the fill unit generates a trace segment to write into the trace cache, if the trace cache already contains a segment starting at the same fetch address, several options are possible. Recall that without path associativity, the trace cache can store only one trace segment starting at a particular address. The first policy blindly overwrites the existing trace segment with the newer one. The second policy keeps the existing version, discarding the newer one. The third policy only writes the newer segment if it contains a different path or is a longer version of the existing segment. Referred to as the *keep-longest* policy, this is the policy used in all the experiments conducted thus far. Figure 7.2 shows the performance of the three policies. For all policies, if a segment contains a newly promoted or demoted branch, then the segment is always written into the cache.

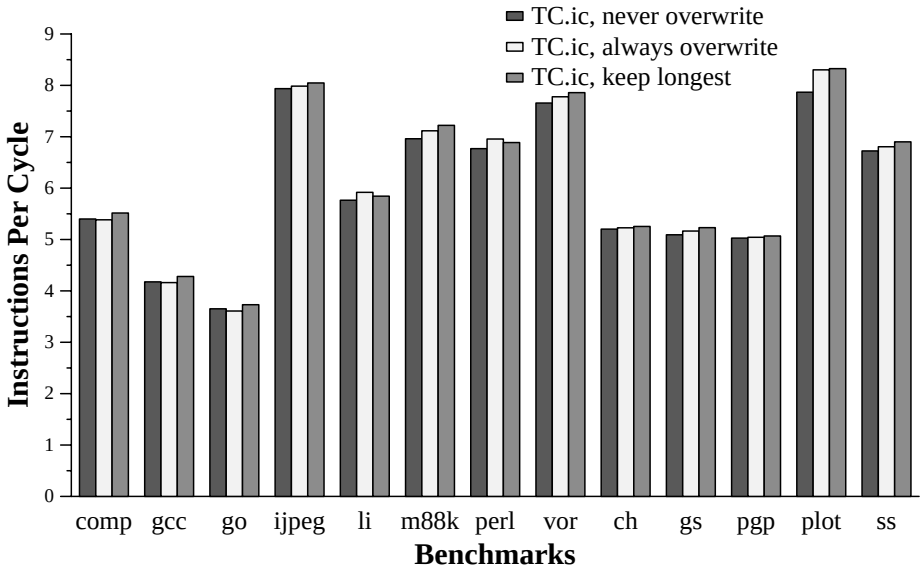

Figure 7.2: The effect of the trace cache write policy on performance.

Over all but two benchmarks (li and perl), the keep-longest policy performs best, and over all but one benchmark (go), the never-overwrite policy performs worst. The never-overwrite policy hinders the adaptability of the trace cache to long term changes in likely trace segments through the program. The keep-longest policy works well because it allows

for this adaptability, while not forcibly removing an otherwise equivalent trace which contains more instructions. Such situations occur because mispredicted branches terminate trace segments.

In order to implement any of these policies, either an extra read port of the trace cache tags is required or the tag information is kept along with each segment after it is fetched. The tag indicates if a trace segment beginning at the same fetch address exists within the cache, and in which element of the set it exists. In order to implement the keep-longest policy, each trace segment's path information also needs to be kept in the tag store (doing so may make other policies such as Partial Matching easier to implement, anyways). By comparing the existing tags and paths with the new segment's tag and path, the write policy logic can determine if the new segment is longer or encodes a different path an existing segment.

Another byproduct of the trace cache write policy is the number of trace segments generated by the fill unit that are actually written to the cache can be lowered. If every generated trace segment is not written, then the trace cache update hardware potentially can be tuned for lower bandwidth. The upper row of Table 7.2 shows the percent of generated trace segments that are written into the trace cache, averaged over all the benchmarks. The lower row show the percentage of total cycles in which a write to the trace cache must be performed. The keep-longest policy attains both high performance while lowering the trace cache write bandwidth requirements.

	Never Overwrite	Always Overwrite	Keep-longest
Percent Written	2.44%	100.00%	6.93%
Percent Write Cycles	1.12%	46.40%	3.13%

Table 7.2: Percentage of trace segments actually written with the three write policies.

7.3 Trace Cache Latency

This experiment involves examining the effect of trace cache latency on overall performance. For this experiment, the latency of the trace cache in the TC.ic configuration is

increased from one cycle to two cycles. In order to accomplish this without halving the fetch bandwidth, the trace cache is pipelined: a new request can be started every cycle, with each trace cache hit taking two cycles to complete.

Starting a request every cycle requires a new target address to be generated every cycle, in the same manner that an instruction cache mechanism requires a Branch Target Buffer (BTB) to generate a taken branch target. For this experiment, a small Trace Target Buffer (TTB) stores the targets of recently accessed trace segments. This structure must have a one cycle latency. At the beginning of the cycle it is accessed with the current fetch address. The fetch address selects an entry within the TTB, with each entry containing the four possible targets of a particular segment. At the end of the cycle, the branch predictions made by the predictor select which of the four targets to use as the next fetch address for the next cycle. Two cycles later, after the trace cache delivers the trace segment corresponding to this fetch, the next fetch address determined from the TTB is confirmed with the trace segment. If the addresses match, then the fetch mechanism proceeds forward. If the addresses do not match, then the fetch mechanism needs to be redirected to the correct target and a two cycle pipeline bubble (called the misfetch penalty) is incurred. There is a subtle difference between the Trace Target Buffer and the Branch Target Buffer. Even though an entry exists in the TTB for a particular fetch address, the next fetch address can be incorrect and a misfetch penalty incurred because the TTB entry and the corresponding trace segment refer to different paths.

Figure 7.3 shows a plot of four different versions of the TC.ic configuration. The baseline is the enhanced TC.ic configuration with a 1 cycle trace cache. The remaining three versions all have a 2 cycle, pipelined trace cache with a trace target buffer of 16KB, 8KB, and 4KB. The percentages above the baseline configuration indicate the drop in performance in going from the baseline to the 2-cycle trace cache with 16KB TTB.

There are two factors that effect performance: an increase in branch resolution time and an increase in misfetch penalty. First, as an extra cycle of latency is added to the trace cache (i.e., from the baseline to 2-cycle trace cache with 16KB TTB), there is a drop in performance due to the increase in branch resolution time which affects performance whenever a branch is mispredicted. Second, as the size of TTB is reduced, more cycles are spent recovering from misfetches.

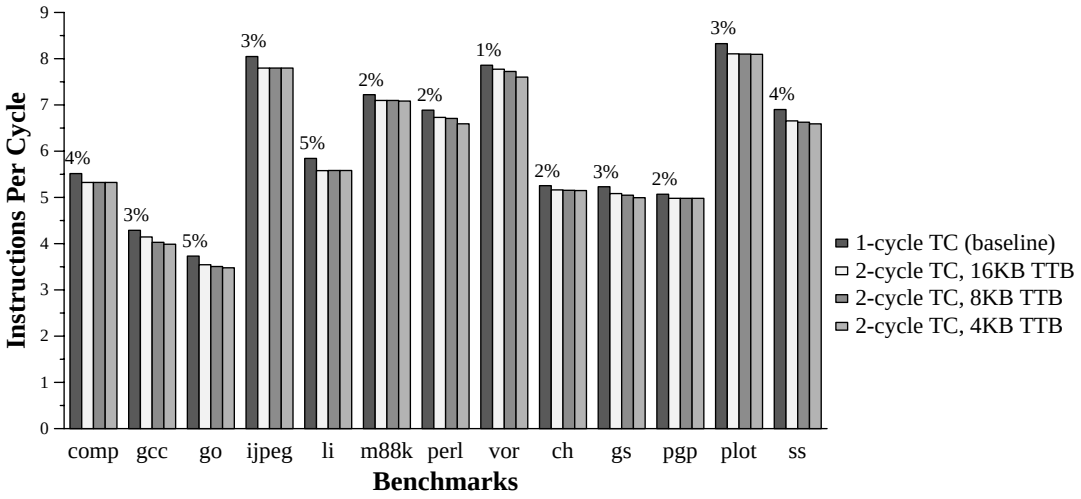

Figure 7.3: The effect of the trace cache latency on the performance of the enhanced TC.ic configuration.

7.4 Evaluating the Compiler Optimizations

In this section, the effect of the compiler code layout optimizations on trace cache performance is measured. As mentioned in Section 4.3, profile-driven compiler optimizations are performed to reduce the occurrence of taken branches. These optimizations have the effect of increasing the instruction bandwidth of the Sequential-Block ICache. The objective of this experiment is to determine whether there is a strong effect on trace cache performance from these optimizations.

Figure 7.4 shows the performance of the Enhanced TC.ic configuration on two versions of each benchmark program. In the first version, the compiler code layout optimizations were enabled (these are the same executables used throughout this dissertation). In the second version, the optimizations are turned off. Note that because the benchmarks have been compiled into essentially different programs which execute a different number of instructions, measuring performance in terms of Instructions Per Cycle is no longer valid. Instead performance is measured by total cycles required for execution. On a similar note, the benchmarks li and ijpeg are omitted from consideration. With the input sets in the benchmark suite, these benchmarks do not execute until completion but rather to the arbitrary limit of 500M instructions. A comparison of these benchmarks is likely to be erroneous since the simulations will terminate at different regions of these programs.

Based on the data presented for the trace cache, these optimizations are helpful, but

not essential.

Figure 7.5 shows the performance of the Sequential-Block ICache with the same two versions of each benchmark program. Here, the compiler optimizations are more of a factor on performance. The programs gcc and m88ksim exhibit a significant increase in performance when the optimizations are enabled. The benchmark gs suffers a loss in performance with the optimizations, attributed to differences in behavior of critical branches between the profiling run and the simulation run.

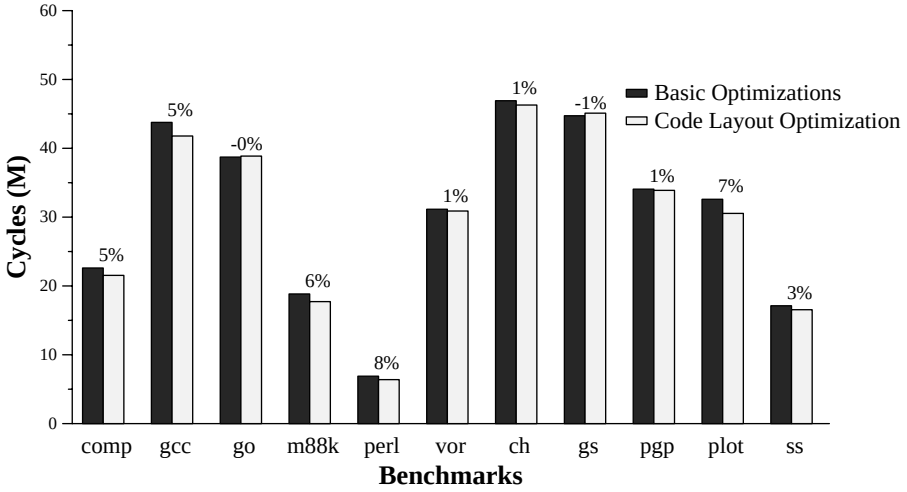

Figure 7.4: The effect of compiler code layout optimizations on TC.ic performance.

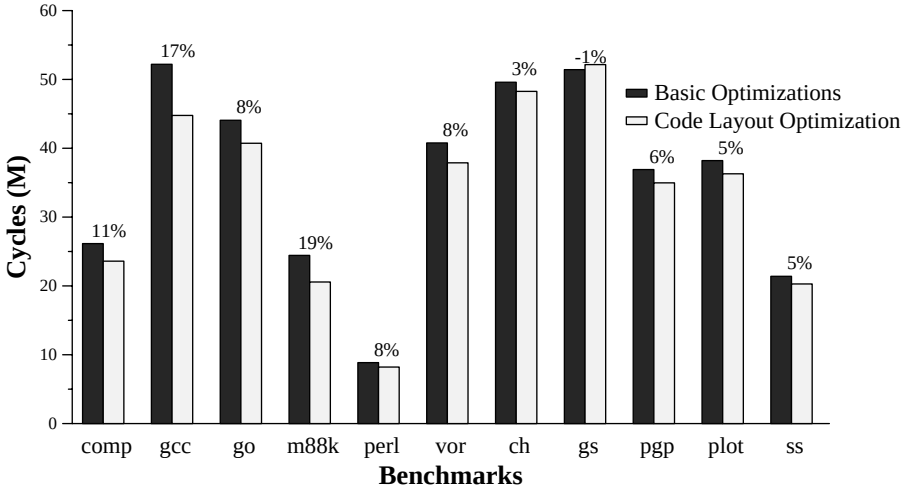

Figure 7.5: The effect of compiler code layout optimizations on Sequential-Block ICache performance.

Table 7.3 shows the growth in number of instructions between taken branches with and

without the compiler optimizations. The compiler is able to increase sequential run length by 35%.

Benchmark	No Code Layout	Code Layout
compress	14.25	26.07
gcc	9.69	13.94
go	11.34	15.05
ijpeg	18.69	31.64
li	8.14	10.85
m88ksim	9.76	13.92
perl	8.91	10.04
vortex	11.32	12.44
chess	19.24	21.94
gs	11.09	10.93
pgp	17.31	24.95
plot	12.41	15.33
ss	9.37	10.41
Average	12.42	16.73

Table 7.3: The increase in average number of instructions per taken branch with and without compiler optimizations.

7.5 Fetch Pipeline Implications

In this section, the two design parameters of the icache are varied. First, the impact of increasing the length of icache pipeline is examined. Second, the impact of converting the supporting icache into a more aggressive Sequential-Block ICache is measured.

7.5.1 Sensitivity to icache path latency

As instructions are fetched from the instruction cache in response to a trace cache miss, they must be aligned, merged, and decoded before they are latched into the renaming stage. Furthermore, if additional processing is pushed into the fill unit, such as instruction routing and placement, then the icache path must include these steps in processing as well.

To simulate the effects of these additional steps, the number of pipeline stages in the icache path is increased from two cycles to three cycles, as presented in Figures 7.6–7.8. The results are not surprising: if the trace cache is large, then the mechanism is insensitive

to icache pipeline length.

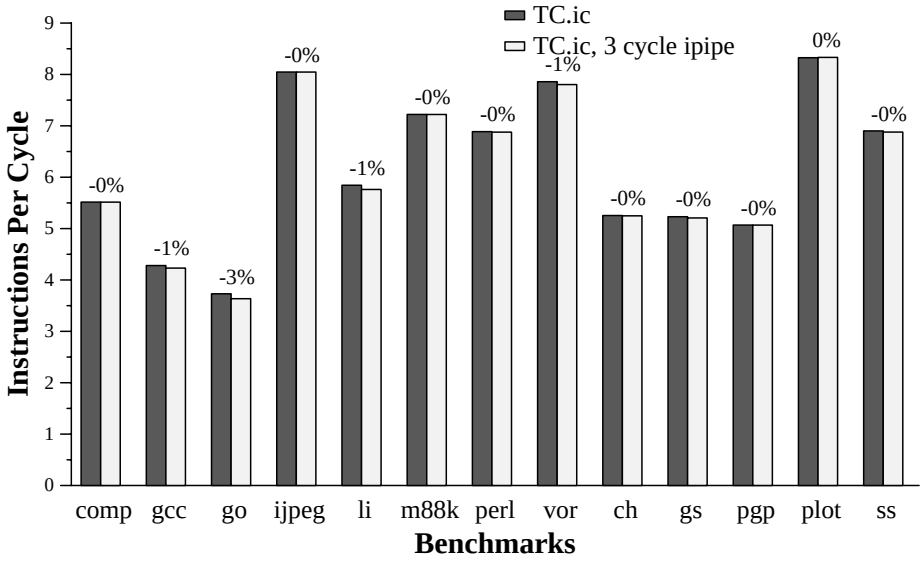

Figure 7.6: TC.ic perf. versus TC.ic with a 3-cycle icache pipeline.

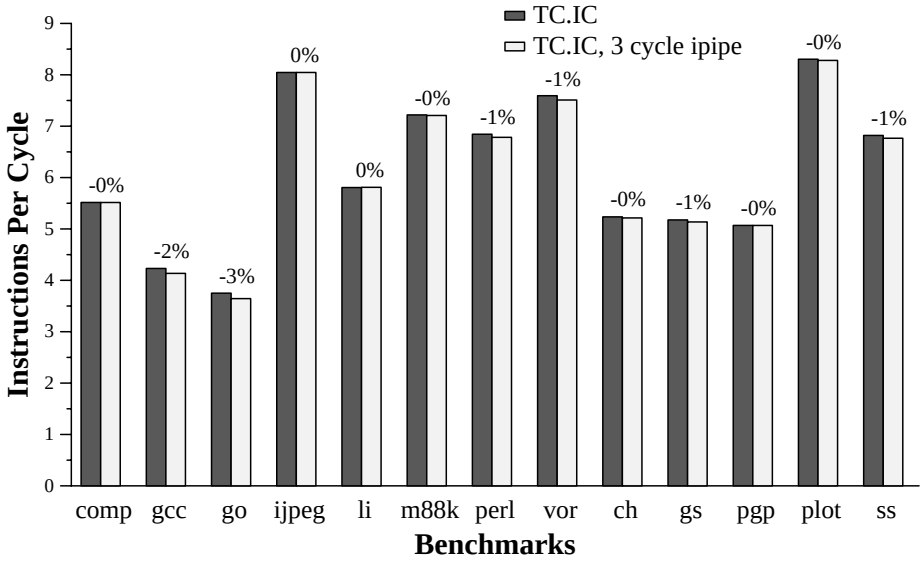

Figure 7.7: TC.IC perf. versus TC.IC with a 3-cycle icache pipeline.

7.5.2 Making the icache path more aggressive

In this experiment, the supporting icache is converted from a simple Single-Block ICache to the more aggressive Sequential-Block ICache. Recall from Section 4.6 that the Sequential-Block ICache delivers up until the first taken branch each cycle. For this experiment, both the trace and instruction cache can provide multiple blocks per cycle.

Figures 7.9- 7.11 show the performance implications for using this scheme on the three

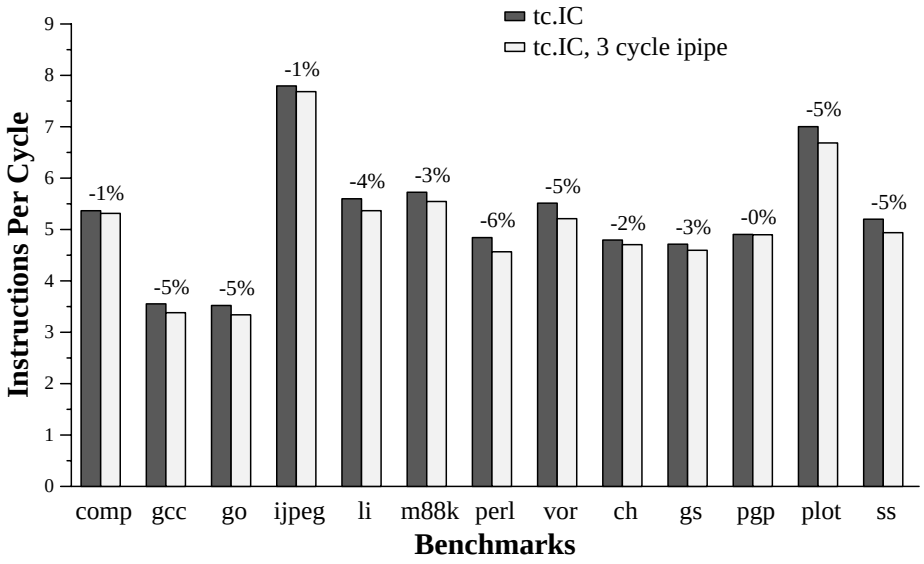

Figure 7.8: tc.IC perf. versus tc.IC with a 3-cycle icache pipeline.

trace cache configurations. The effect of making the icache more aggressive for the TC.ic configuration is negligible, since the icache services a small fraction of fetches. As the relative size of the icache grows, the importance of making it more aggressive grows. It is a significant factor on performance for the tc.IC configuration.

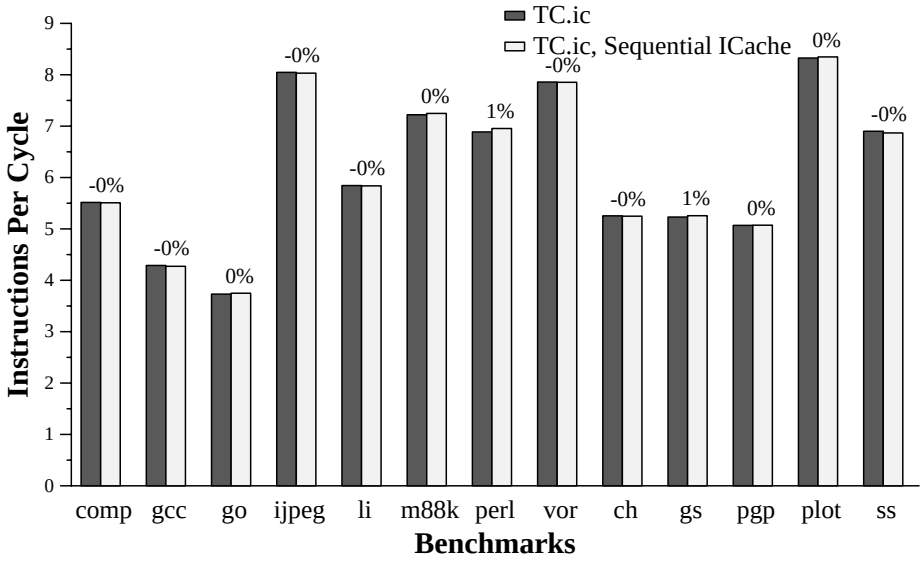

Figure 7.9: TC.ic perf. versus TC.ic with a Sequential-Block ICache.

7.5.3 Storing only non-sequential segments in the trace cache

Using the Sequential-Block ICache with the trace cache opens up another interesting possibility. The trace cache can be used to store only non-sequential trace segments, and

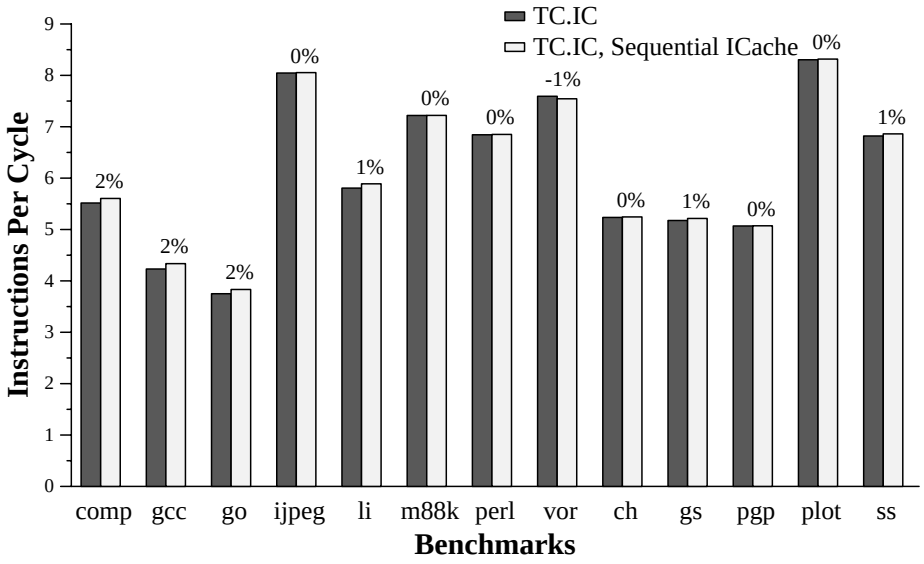

Figure 7.10: TC.IC perf. versus TC.IC with a Sequential-Block ICache.

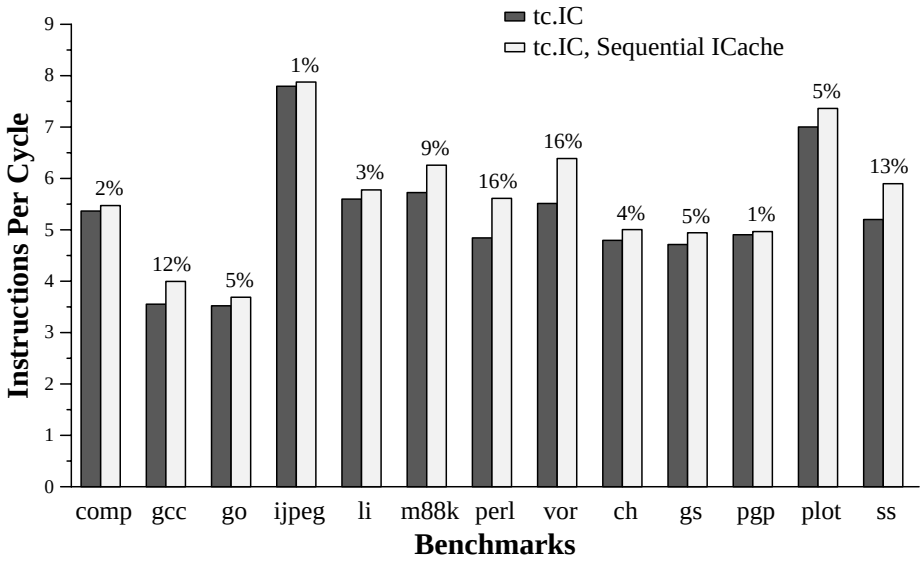

Figure 7.11: tc.IC perf. versus tc.IC with a Sequential-Block ICache.

the icache can be used to deliver sequential runs. Since the trace cache and instruction cache would both, presumably, participate equally in instruction supply, the TC.IC configuration makes the most sense for this strategy. Dividing the caching between sequential and non-sequential provides a capability similar to path associativity (see Section 6.3), as the instruction caching structures can contain the sequential path and one non-sequential path emanating from a particular instruction.

Precisely stated, the trace cache only accepts a trace segment from the fill unit if the segment contains an internal taken branch (conditional or otherwise) or contains a promoted

branch. Each fetch request accesses both caches and if both caches respond with a hit, then a selection must be made using the branch predictions on which of the fetches provides more instructions. This complex selection is likely to lengthen the cache access time.

Furthermore, the trace cache contains decoded instructions, thus eliminating the decoding step between the fetch of a packet from the trace cache and its subsequent issue into the instruction window. On this basis, there is more value in fetching the instructions in the trace cache than from the instruction cache. For this experiment, this extra pipeline stage on the icache path is eliminated. Here the assumption is made that the icache, like the trace cache, can supply decoded instructions.

Figure 7.12 shows the performance between two TC.IC configurations. Both configurations have equal length trace cache and icache pipelines (i.e., the icache contains decoded instructions) and the packet selection logic is assumed to not lengthen the cache access path. In the configuration labeled non-sequential, the trace cache only stores non-sequential traces (as described above). The benchmarks compress, go, and li suffer a slight performance loss due to a decrease in fetch rate and an increase in branch mispredictions.

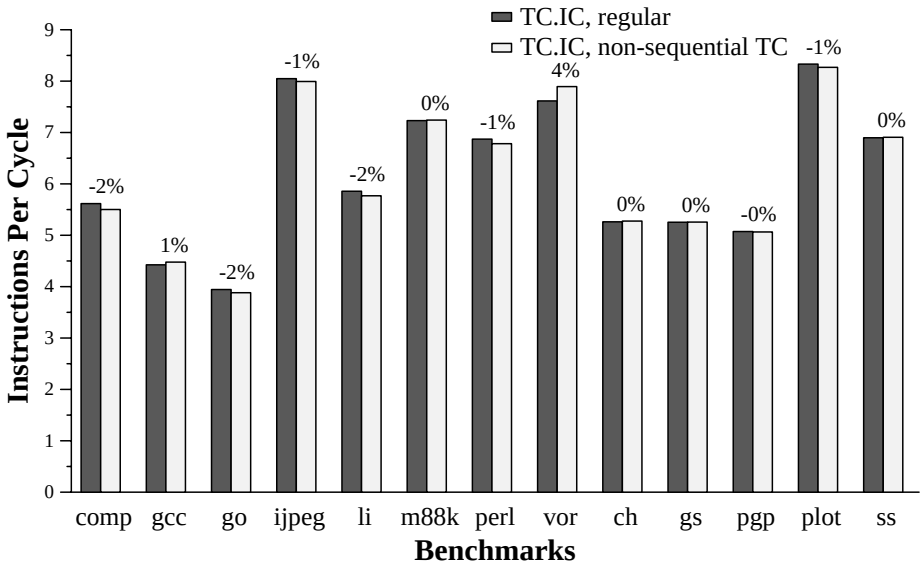

Figure 7.12: Performance of a trace cache which only stores non-sequential segments with a Sequential Block ICache versus a standard TC.IC configuration.

While the performance of this configuration measured in IPC is not substantial relative to the baseline, there are elements of it that make it appealing. Ignoring the issues of longer access time and the uneven pipeline lengths for the moment, the non-sequential trace cache

demonstrates a lower fetch miss rate than the baseline. The average fetch request misses per 1000 instructions are listed in Table 7.4. There is a 10% drop. Effective fetch rate and branch misprediction rates are slightly lower, enough to account for the slight loss in performance for some benchmarks. The large cache sizes and small applications sizes used in this dissertation may mask the effects of this. Smaller caches and larger applications may benefit from such split caching.

	Baseline	Non-Sequential
Ave Miss Rate	0.49	0.44

Table 7.4: Fetch request miss rates for the non-sequential trace cache.

7.6 Fill Unit Configuration

In this section, two basic fill unit experiments are conducted. The first experiment deals with the placement of the fill unit in the instruction processing path: should the fill unit collect blocks of instructions as they retire or as they are issued? The second experiment measures the impact of fill unit latency on the overall performance of the processor.

7.6.1 Block collection at retire or issue

The fill unit collects blocks of instructions as they are processed and produces segments to store into the trace cache. The fill unit can collect these blocks at any point in the processor pipeline. The objective of this experiment is to determine whether the blocks should be collected as instructions are issued into the instruction window or when they are retired.

Figures 7.13–7.15 shows that the differences in performance between the two schemes are insignificant. A fill unit collecting instructions at issue time generates more traffic to the trace cache because segments are collected while the processor is on a wrong execution path. In some cases this prefetches useful segments, but in other cases it evicts useful segments from the trace cache.

A fill unit that collects at retirement time only writes segments from the correct execution path to the trace cache. However, it suffers from an increased latency between the initial fetch of a block and its collection into a segment and subsequent storage into the trace

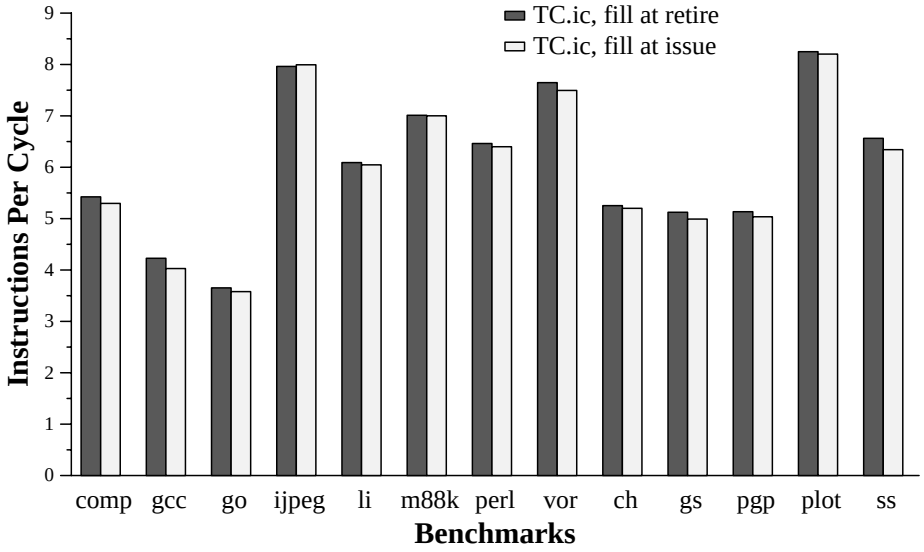

Figure 7.13: The TC.ic performance with fill at retire versus fill at issue.

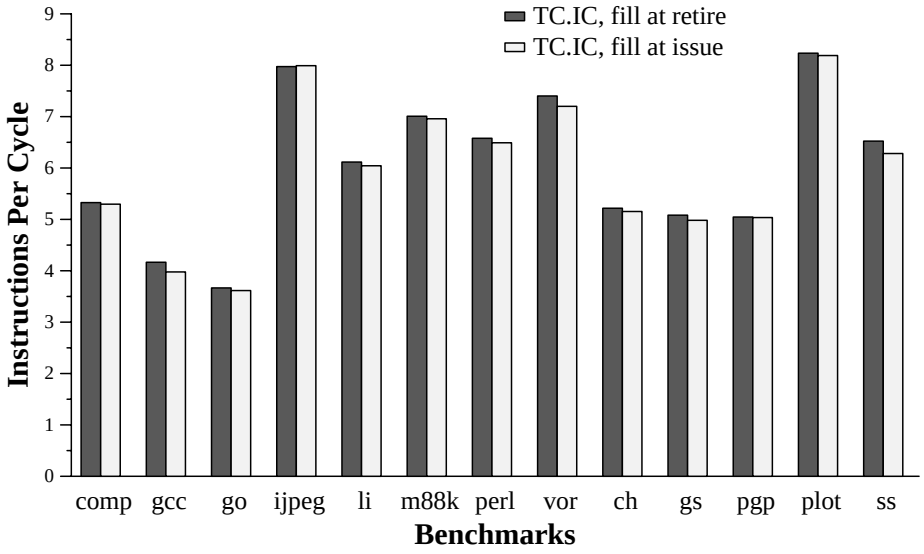

Figure 7.14: The TC.IC performance with fill at retire versus fill at issue.

cache. While this can potentially impact the first few iterations of a tight loop, which will be fetched from the instruction cache until the first iteration retires, the next experiment will demonstrate that this is not an important performance factor.

Although a fill unit that collects at retire time has a slight advantage over one that collect at issue time, collecting segments at retirement time requires a slightly higher hardware investment. Implementing an issue time fill unit is straight-forward, requiring blocks to be latched into the node tables and fill unit concurrently. For retire time collection either the fill unit must contain enough storage to maintain a copy of the instructions in the execution window or blocks of instructions must be driven from the node table into the fill unit as

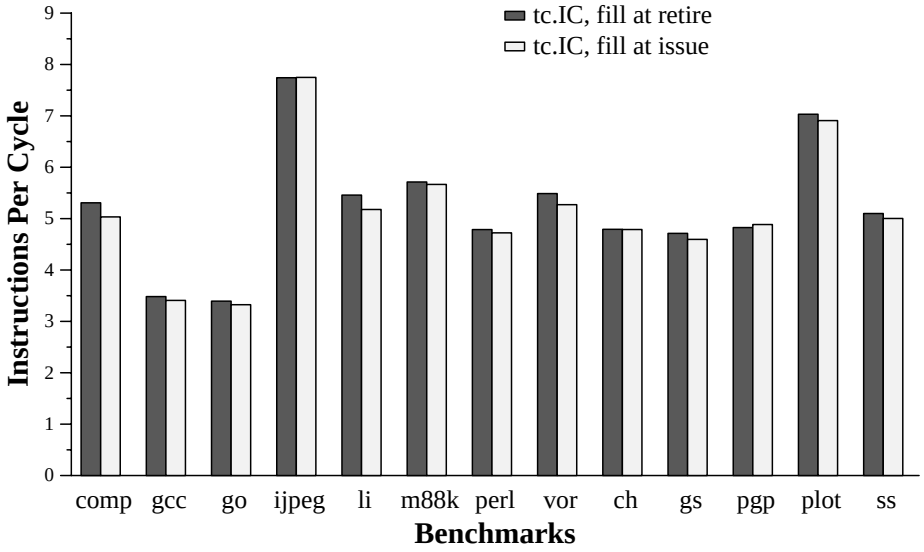

Figure 7.15: The tc.IC performance with fill at retire versus fill at issue.

they retire. In the former case, checkpoints are added to the fill unit as they are issued but are only eligible for merging after they retire. In the latter case routing must be added from the node table to the fill unit so that the checkpoints may be entered as they retire.

7.6.2 Fill unit latency

One of the major advantages of the trace cache mechanism is that operations can be moved off the critical processing path of the machine and into the fill unit path where latency is an insignificant factor on performance. This experiment demonstrates that this latency is indeed inconsequential. Presented in Figures 7.16 is the performance of the TC.ic configuration as the latency of the fill unit is increased up to 10 cycles. There is a very minor effect on performance. These results may seem counter-intuitive. There are two reasons for this behavior. Most importantly, this latency is incurred after the initial demand for the instructions has been satisfied. Therefore a penalty is only paid if the execution path loops back to these instructions before the segment has been written to the trace cache. Furthermore, in the process of satisfying the original demand request, the blocks are brought at least into the instruction cache. Therefore any subsequent requests that occur before the full segment has been written into the trace cache can still be satisfied by the instruction cache. For some benchmarks (e.g., m88ksim and chess), a longer fill unit latency results in a slightly higher performance. A longer latency sometimes delays the update of a useful trace cache segment with a less useful one.

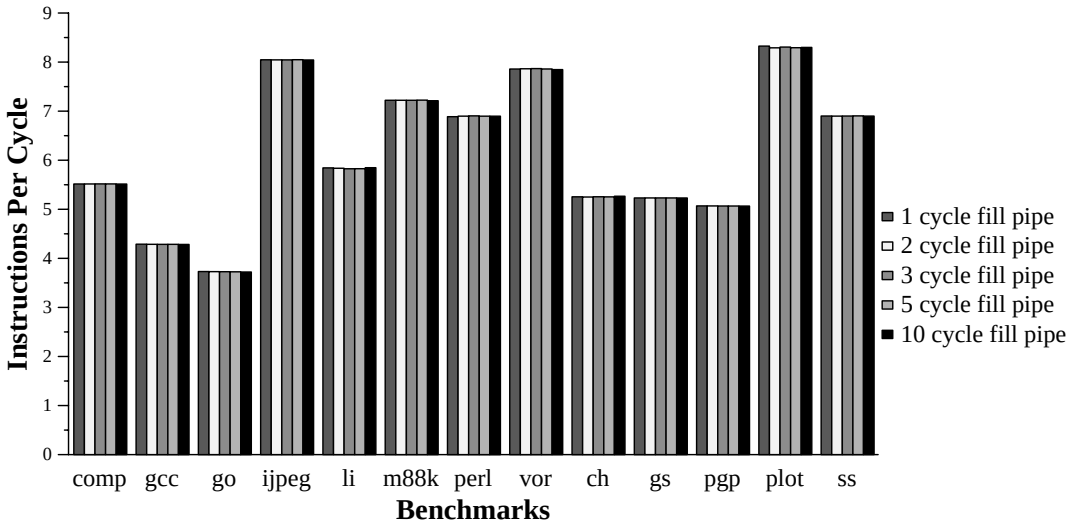

Figure 7.16: The TC.ic performance with various fill unit pipeline lengths.

7.7 Branch Predictor Configurations

The objective of this section is to show the sensitivity of the enhanced trace cache mechanism to simple changes to the branch prediction mechanism. The first experiment examines several Pattern History Table (PHT) entry configurations. The second experiment shows the effect on overall performance as the predictor is (artificially) made more accurate.

Since branch prediction is such a large topic, a comprehensive examination of multiple branch predictors for use with trace caches would be beyond the scope of this dissertation. Instead, some simple variations in predictor design are examined to demonstrate the mechanisms sensitivity to branch predictor performance.

7.7.1 PHT organization

With global-history based two-level branch prediction, the first level of branch history contains the recent outcomes of all branches. This first-level history is used to index into the Pattern History Table, which stores the likely outcome when a particular pattern in the first level is encountered. It has been demonstrated that for most global-based schemes, saturating counters (e.g., 2-bit counters) are effective at maintaining these likely outcomes [47]. For multiple branch prediction, these PHT entries can be augmented to supply more than one prediction for each PHT access. In this section, the performance of three different types of PHT entries are compared. The first is the baseline predictor used for the previous experiments in this dissertation. The second is a PHT entry configuration proposed in an older

study [37]. The third is a likely path scheme proposed by Menezes et al. [33]. These PHT schemes are three that have been proposed in the literature for multiple branch prediction.

As described in Section 2.4, the baseline predictor used for this study is a two-level predictor capable of predicting three branches each cycle. Each PHT entry consists of three 2-bit counters, each counter providing a prediction for a single branch. Since Branch Promotion (see Section 6.6) significantly decreases the occurrences of fetches which require two or three predictions, the predictor can be optimized by decoupling the three counters within each PHT entry into three separate PHTs. The first PHT contains the 2-bit counter used to predict the first branch, the second, the second branch, and so forth. The size of the first PHT is made much larger than the second, and the second larger than the third.

The second scheme expands the notion of the three counter scheme. Each entry consists of seven 2-bit counters: the first counter supplies the prediction for the first branch and selects one of two counters to supply the prediction for the second branch. The first two predictions are then used to select one of four counters to provide the prediction for the third branch. A diagram for this scheme is shown in Figure 7.17.

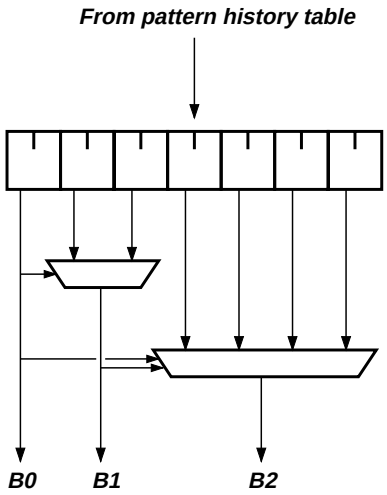

Figure 7.17: The multiple branch predictor supplies 3 predictions per cycle using seven 2-bit counters per PHT entry.

The final scheme, the likely-path scheme, was presented by Menezes et al [33]. In this scheme, each PHT entry holds the most likely path through a program subgraph containing three branches. In this scheme, the PHT entries are four bits wide: three bits to encode the likely path (eight paths are possible) and a fourth bit to record the likeliness of this path.

Figure 7.18 displays the overall performance and Figure 7.19 displays the branch mis-

prediction rates for each of the three schemes. The optimized 3-counter is allocated 24KB of storage, the 7-counter scheme is allocated 32KB of storage, and the likely path scheme 32KB. All were measured on the enhanced TC.ic configuration. The changes in performance due to the different PHT schemes is insignificant. Generally, the 7-counter scheme suffers the highest misprediction rate, thus has the lowest overall performance than the other two schemes because of it's poor allocation of counters among the branches in a packet. The 3-counter scheme performs well at the lowest hardware cost (in terms of bits of storage).

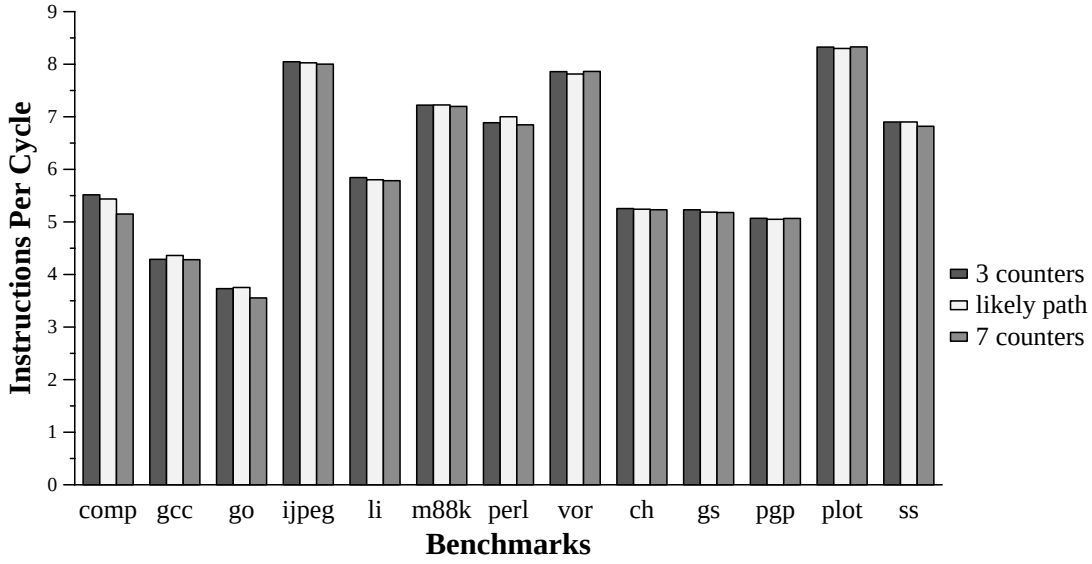

Figure 7.18: The performance of the various PHT entry schemes.

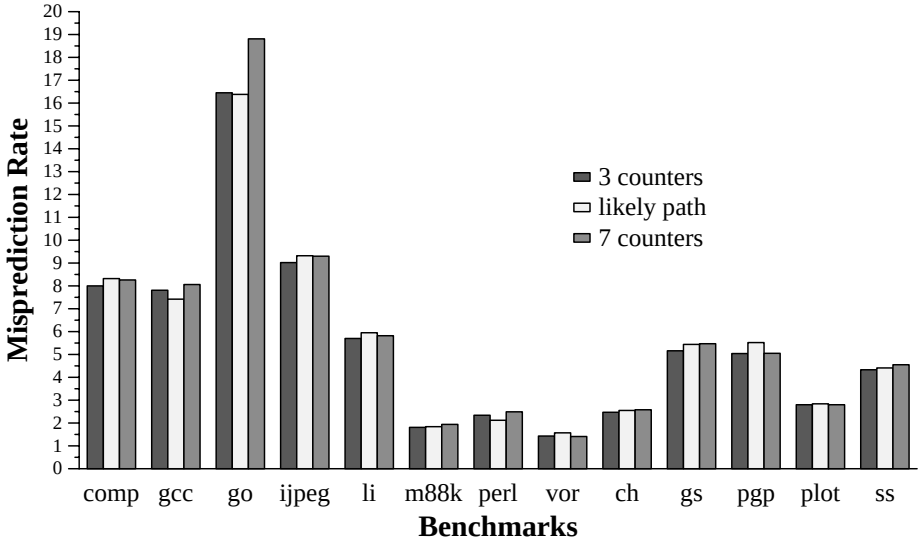

Figure 7.19: The branch misprediction rate for the various PHT entry schemes.

7.7.2 The Effect of Branch Prediction Accuracy

In this experiment, the effect of branch prediction accuracy on overall performance is measured on the gcc benchmark running on the enhanced TC.ic configuration. Figure 7.20 is a plot of data points which demonstrates the relationship between prediction accuracy and performance. The curve is not flat. As the predictor becomes more accurate, the absolute performance grows more rapidly.

Five of the data points in this graph correspond to real predictors. They are denoted on the graph as solid circles, and they correspond to various sizes (6KB, 12KB, 24KB, 48KB, and 96KB) of the 3-counter predictor optimized for branch promotion. To generate the other points on this plot, these real predictors were synthetically made more accurate by switching the prediction on randomly chosen incorrect predictions, i.e., incorrect predictions were randomly made correct. The probability of switching a prediction is a parameter to the simulation. When this probability is low, the synthetic predictor performs similar to the underlying real predictor. When high, the synthetic predictor performs closer to the ideal. With this methodology, the synthetically generated data points are based on a real predictor. To generate a synthetic data point, three parameters are required: the underlying real predictor, the probability of converting a real misprediction into a correct one, and a seed for the random number generator. Various values of probability and seeds were used to generate the points presented in the figure.

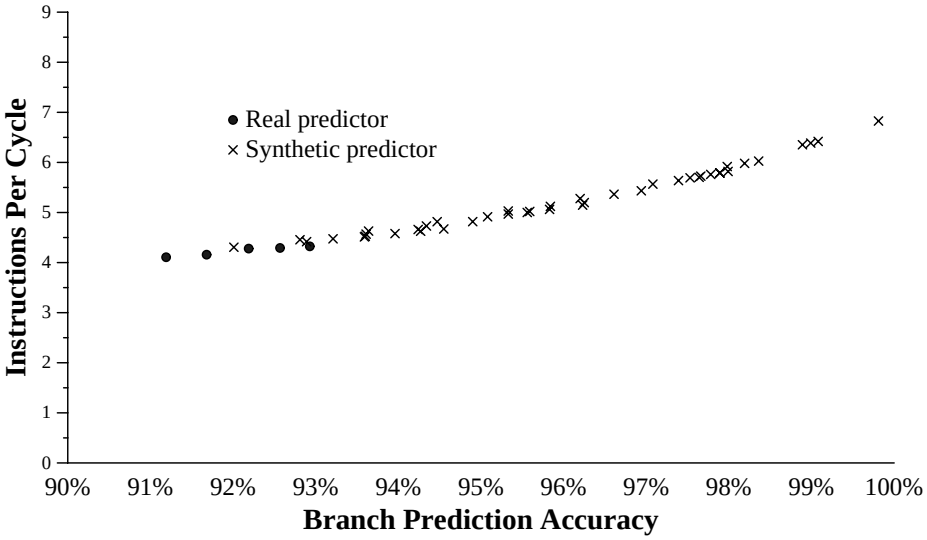

Figure 7.20: The effect of branch misprediction rate on overall performance.

CHAPTER 8

Analysis

8.1 Overview

A global analysis of the trace cache mechanism is presented in this chapter. The first section deals with the relationship between fetch rate and branch resolution time, bringing to light the interplay between various bottlenecks in superscalar processors. The second section deals with instruction duplication within the trace cache. In this section, an experimental quantification of duplication is presented, along with methods of reducing it. The final section discusses the large Execution Atomic Units possible with the trace cache.

8.2 The Relationship Between Fetch Rate and Branch Resolution Time

Figure 8.1 shows an interesting trend. In this figure, the SPECint95 benchmarks gcc and go are simulated on three different hardware configurations. The only difference between the hardware configurations is the width of the fetch mechanism, which is varied from four to eight to sixteen. All other parameters are kept constant: the size of the trace and instruction caches, branch predictor, BTB, and all parameters of the execution engine (which is fixed at 16-wide issue and execute) remain unchanged. Changing the fetch width involves changing the size of each trace cache line from four to eight to sixteen instructions.

As the fetch width is increased, the effective fetch rate increases, and the number of execution cycles spent fetching the dynamic program decreases. Since all other parameters are kept constant, one would expect the number of cycles lost due to cache misses and

branch mispredictions to remain, for the most part, constant. However, the data plotted in the figure demonstrates that the number of cycles lost due to branch misses increases substantially as the fetch rate is increased.

Since the trace cache involves a minor tradeoff between higher fetch rate for slightly higher branch misprediction and cache miss rates, part of this increase in branch miss cycles is due to a slightly higher misprediction rate. However, the benchmark gcc suffers an 8% increase in the number of mispredicted branches, but an 82% increase in the number of cycles lost to branch mispredictions in going from 4-wide fetch to 16-wide fetch. Similarly, the benchmark go suffers a 5% increase in the number of mispredicted branches but a 70% increase in the number of cycles lost to mispredictions.

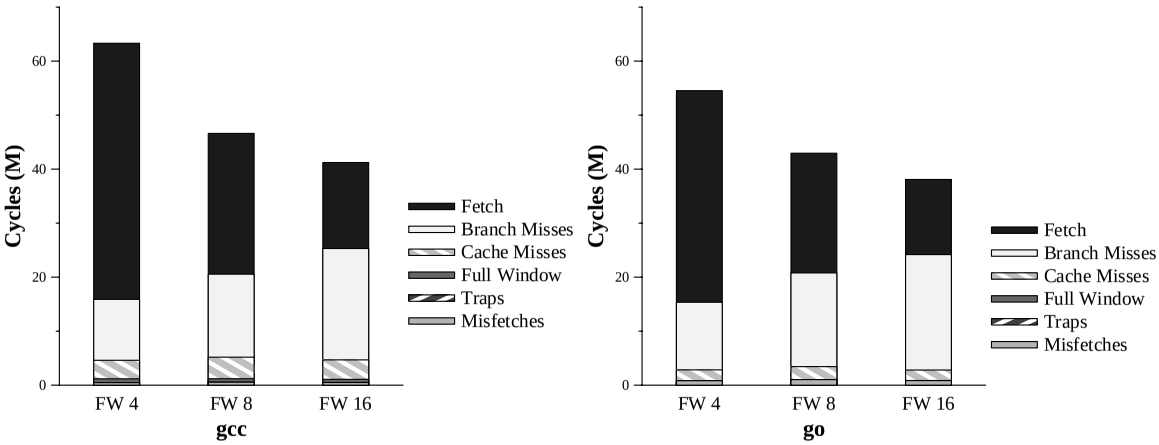

Figure 8.1: Cycle breakdown of the SPECint95 benchmarks go and gcc at fetch width of 4, 8, and 16

Since the number of cycles lost to branch mispredictions is a product of the number of branches mispredicted and the time needed to *resolve* each one, the conclusion is that the branch resolution time per mispredicted branch is increasing. Figure 8.2 demonstrates the phenomenon: the horizontal axis marks the effective fetch rate and the vertical axis marks the average number of cycles needed to resolve a mispredicted branch averaged over all the SPECint95 benchmarks.

The three curves represent three execution engines, each capable of issuing 16 instructions each cycle. The Conservative engine has a 2-cycle 64KB data cache, uses a simple memory ordering model where older stores with unknown addresses block all younger memory operations, and has a 1-cycle communication delay for forwarding a value from one

cluster of execution units to another. The 16 uniform execution units are clustered in groups of four. The Aggressive engine is similar except it has a 1-cycle data cache, uses dependence prediction (modelled as perfect) to allow memory operations with no in-flight dependencies to proceed, and has no extra communication latency for bypassing values from one execution unit to another. The Ideal engine has infinite execution units and a perfect data cache and uses perfect memory ordering.

Each point represents a single configuration of frontend (i.e., trace cache fetch width) and backend (i.e., Conservative, Aggressive, Ideal). The horizontal axis denotes the harmonic mean effective fetch rate on the SPECint95 benchmarks and the vertical axis the arithmetic mean branch resolution time.

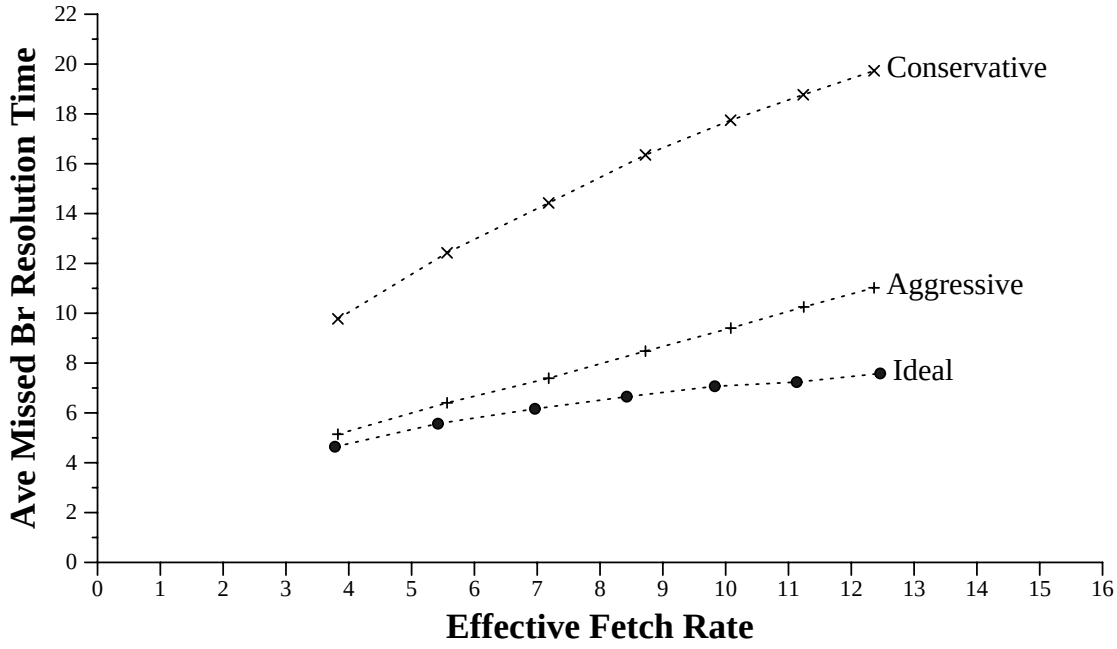

Figure 8.2: The effect of the increasing the effective fetch rate on mispredicted branch resolution time.

All three curves demonstrate that average branch resolution time increases as the effective fetch rate is increased. There are two reasons for this: execution bandwidth and the amount of exploitable instruction level parallelism within the applications. The influence of execution bandwidth is demonstrated by the sharp contrast in magnitude and slope between the three curves in Figure 8.2. The Conservative engine adds extra artificial delays in the processing of instructions, as instructions are forced to wait for values to be communicated, for cache ports, for execution units and distribution busses. The Aggressive Engine

has fewer of these limitations, but artificial delays are incurred nonetheless. For the Ideal engine, the increase is due purely to the fact that the instruction level parallelism within the application delays the branch's execution.

To further quantify the limitation due to instruction level parallelism, the Ideal engine was augmented to measure the depth of the dependence tree at fetch time for each branch which was mispredicted. The depth of the dependence tree represents the longest chain of instructions which remain to be executed before the branch can execute. If the depth of a particular branch is, say, seven, then the earliest the branch can execute is seven cycles after it is inserted into the instruction window. Since some instructions take more than a cycle to execute, the depth (which is a count in number of instructions) is a lower bound. Figure 8.3 shows the increase in average depth for mispredicted branches as the effective fetch rate is increased. Each data point represents an average of all SPECint95 benchmarks.

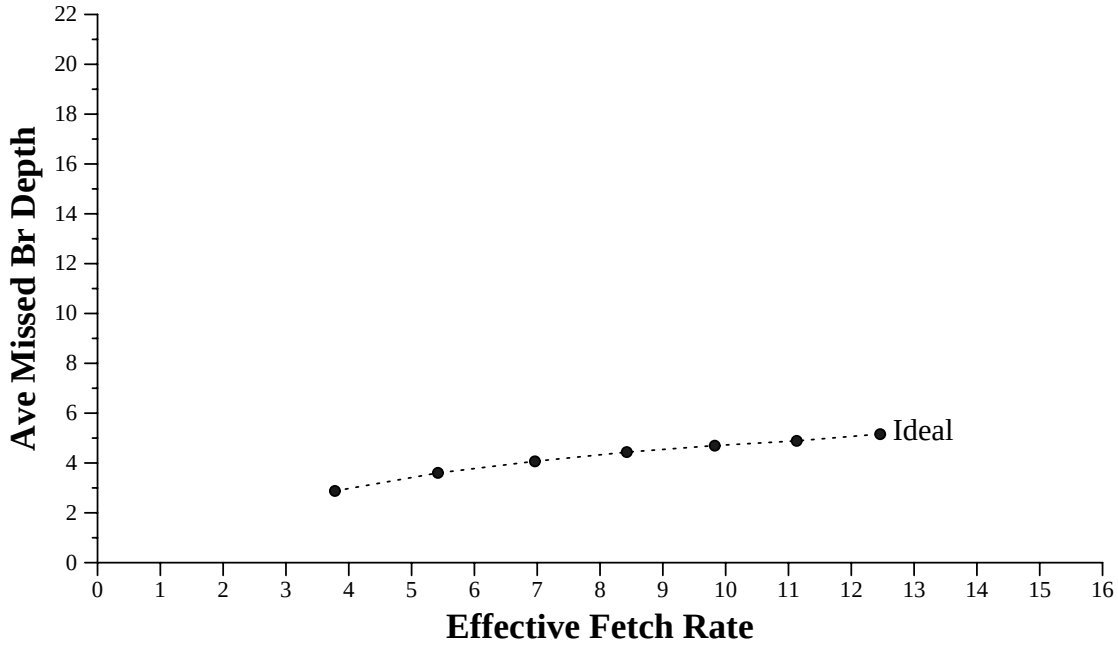

Figure 8.3: The effect of the increasing the effective fetch rate on mispredicted branch depth.

The data presented in this section highlight the importance of branch prediction in light of increased effective fetch rates. As the partial fetch bottleneck is mitigated, the branch bottleneck becomes aggravated. Because of limitations in instruction level parallelism, higher fetch rates cause branches to wait longer before execution (Figure 8.3). Furthermore, artificial execution delays cause each of these steps to take additional cycles (Figure 8.2).

8.3 Duplication in the Trace Cache

Because of the nature in which trace segments are created, a single instruction can exist in several different trace cache lines simultaneously. Some of these duplicately cached copies may contribute to the trace cache's bottom line of delivering higher instruction bandwidth, and some may not. On this basis, instruction duplication can be divided into two categories: replication and redundancy. Instruction replications are useful copies, and the have a positive impact despite increasing trace cache storage contention. Redundant copies, on the other hand, increase cache contention without contributing to performance. The objective of this section is to differentiate between, measure, and analyze replication and redundancy in the trace cache. Based on this analysis, several modifications are made to the trace segment finalization strategy to lower instruction redundancy.

8.3.1 How does this duplication occur?

Because of the dynamic nature of trace creation, instructions may get cached in multiple locations. It is this duplication that partly contributes to the trace cache's ability to deliver instructions at a high rate. However, the resulting inefficient usage of the cache space results in a higher miss rate for the trace cache than an instruction cache of the same size.

Figure 8.4 helps illustrate the point. It shows a simple loop composed of three blocks. If blocks are treated atomically (i.e., no trace packing is performed), the fill unit can potentially create three combinations: AB, CA, BC, all three of which can be simultaneously resident in the trace cache. Generally speaking, duplication is caused by blocks within a program where multiple control paths merge, such as block A in the figure.

With all these combinations in place, the trace cache can deliver, in the steady state, two iterations every three cycles. On the negative side, the extra segments may displace other useful cache lines. If duplication were inhibited by only allowing instructions to exist in one trace segment (i.e., only the segment AB and segment C are formed in the steady state), then only one iteration can be delivered every three cycles but the loop is expressed with two fewer trace segments. The tradeoff here is between higher bandwidth from fetching fuller segments versus bandwidth losses due to increased misses in the trace cache.

Trace packing, while boosting the fetch rate even further, also boosts duplication. The example loop shown in figure 8.4 would result in six unique segments being created (with

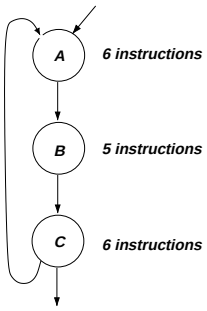

Figure 8.4: A loop composed of 3 fetch blocks.

unregulated Trace Packing), but these six segments would allow the loop to issue at a rate of 2.5 iterations every three cycles. [1] Packing small, tight loops with many iterations is likely to be a win because the cost of displacing cache lines is offset by the burst in instruction fetch bandwidth.

There is another form of this duplication which is entirely helpful. If several iterations of a tight loop are peeled into a single trace segment (segment AAA, for example), then the copies of the instructions only contribute to a higher fetch rate without causing extra trace cache lines to be generated.

Duplication can be divided into two categories: useful copies and useless copies. Useful duplication, called replication, contributes to the trace cache's bottom line of delivering high effective fetch rates. Useless duplication, called redundancy, simply occupies cache space without significantly boosting fetch rate. To improve performance of the trace cache mechanism, redundancy must be eliminated.

8.3.2 Measuring duplication

The first objective is to measure average instruction duplication. This is done by scanning the trace cache with every instruction fetched and counting the number of other trace segments that also contain that instruction. [2] Figure 8.5 shows the average number of copies of an average instruction as it is fetched. This data was collect on the experimental benchmark set on three variations of the TC.ic configuration. The first variation uses no Trace Packing. The second uses cost-regulated Trace Packing (See Section 6.7). The final varia-

[1] The six segments are $A_6B_5C_5$, $C_1A_6B_5$, $C_6A_6B_4$, $B_1C_6A_6$, $B_5C_6A_5$, $A_1B_5C_6$. The subscripts denote the number of instructions of each block included in the segment. Notice that even with trace packing, no instructions beyond the third branch are added to the pending segment

[2] The experimental setup allowed for an efficient way to detect the number of copies of each instruction fetched.

tion uses unregulated Trace Packing, i.e., all trace segments not containing three branches, or a return or indirect branch instruction are packed with 16 instructions. The average duplication count per instruction is significantly higher with unregulated trace packing.

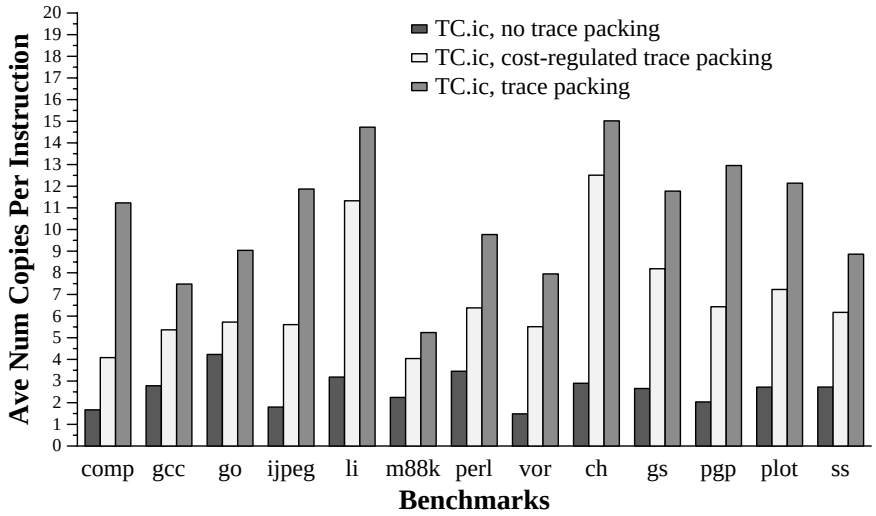

Figure 8.5: The average number of copies of an instruction fetched from the trace cache for three trace cache packing strategies.

Another side effect of duplication is an increase in the number of unique trace segments dynamically generated by the fill unit. A trace segment is considered unique based on its starting address and the sequence of instructions it contains. Table 8.1 lists the number of unique traces generated by the Enhanced TC.ic configuration (using cost-regulated trace packing) for each benchmark.

8.3.3 Analyzing duplication

The duplication measured in the previous section is presented in Figures 8.6 and 8.7 as a distribution of the individual duplications. As before, as an instruction is fetched from the trace cache, a probe is performed to determine the number of copies of that instruction which currently exist (i.e., the number of other cached trace segments in which that instruction currently exists). The graphs in Figures 8.6 and 8.7 are histograms where each instruction fetched from the trace cache is binned by duplication factor. For example, an instruction fetched at, say, cycle 3102, having 3 total copies residing in the cache would increment bin number 3. The sum of all the bins results in the percentage of dynamic instruction which are fetched from the trace cache.

Benchmark	Number of Unique Traces
compress	1408
gcc	115904
go	52362
ijpeg	5110
li	4600
m88ksim	4725
perl	6172
vortex	26258
chess	18866
gs	47041
pgp	8145
plot	10065
ss	11479

Table 8.1: The number of unique traces created by the Enhanced TC.ic configuration.

There are two notable outliers: li and chess. Both of these benchmarks have a very high amount of duplication and both have distributions where the most frequent duplication factors are 16 or greater. The other benchmarks show more gradual duplication, where the most smaller duplication factors are more frequent and the larger ones less frequent.

The next objective is to divide duplication into replication (useful duplication) and redundancy (useless duplication). To judge whether a copy of an instruction is useful, a methodology similar to profiling is used: a program is simulated on the Enhanced TC.ic configuration. During the simulation, a list of all trace segments generated and the number of times each is requested (on-path) is maintained. At the end of the simulation, this entire list is dumped into a file. Next, the same program with the same input set is simulated again on the same configuration. This time, the trace segment list dumped to file from the previous simulation is read into an internal structure. The new simulation thereby has full knowledge of the ending statistics of all trace segments generated as the program executes. Now, as the program is simulated, duplication is measured as before. Each duplication measurement is cross-checked with the trace segment list to determine if a particular copy of an instruction is in a trace segment that will be read more than a *threshold* number of times. If so, the copy is considered a useful replication. Duplicate copies which are in trace segments accessed fewer then *threshold* times are considered redundant.

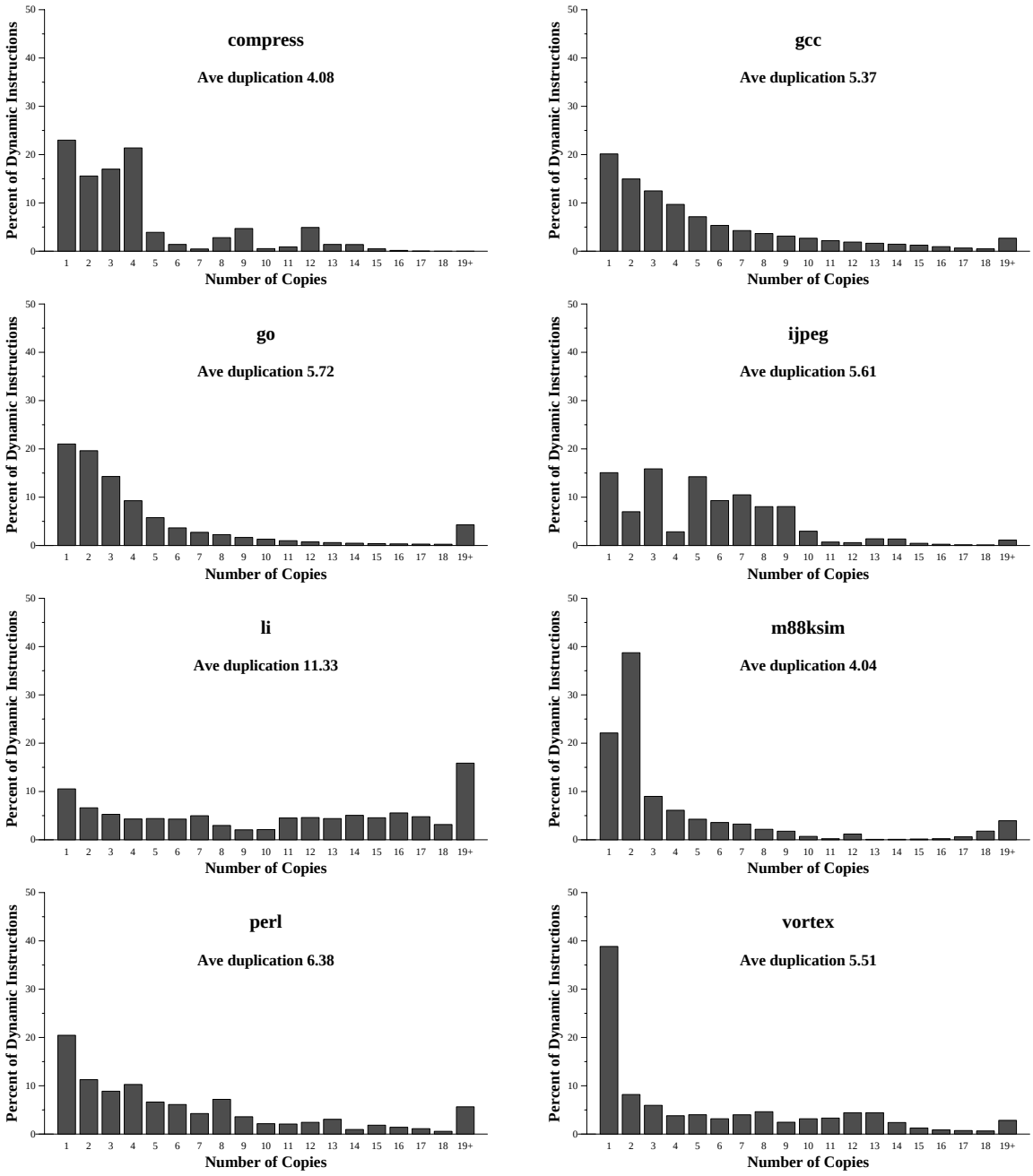

Figure 8.6: Distribution of amount of duplication for the SPECint95 benchmarks on the Enhanced TC.ic configuration (cost-regulated trace packing). The x-axis represents the number of copies resident in the trace cache when an instruction is fetched.

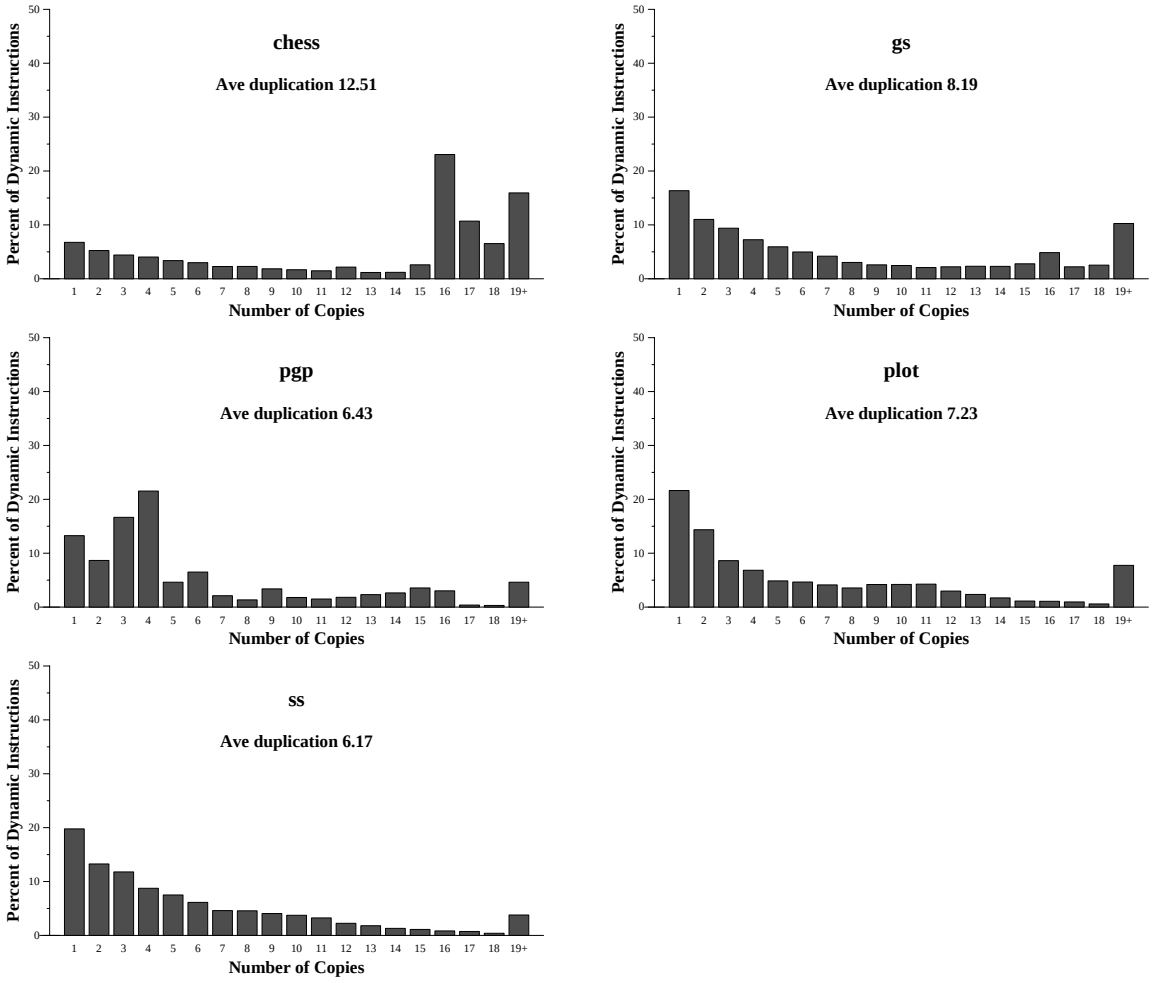

Figure 8.7: Distribution of amount of duplication for the UNIX benchmarks on the Enhanced TC.ic configuration (cost-regulated trace packing). The x-axis represents the number of copies resident in the trace cache when an instruction is fetched.

Figures 8.8–8.11 show for each benchmark the replication distribution for all instructions fetched from the trace cache. The left-hand figure is a histogram for *threshold*=0 and the right-hand figure is a histogram for *threshold*=10. Notice for *threshold*=10, there are instructions fetched with a zero replication count: here, fetches are made from trace segments which themselves are categorized as redundant, i.e., they will be accessed fewer than 10 times in total.

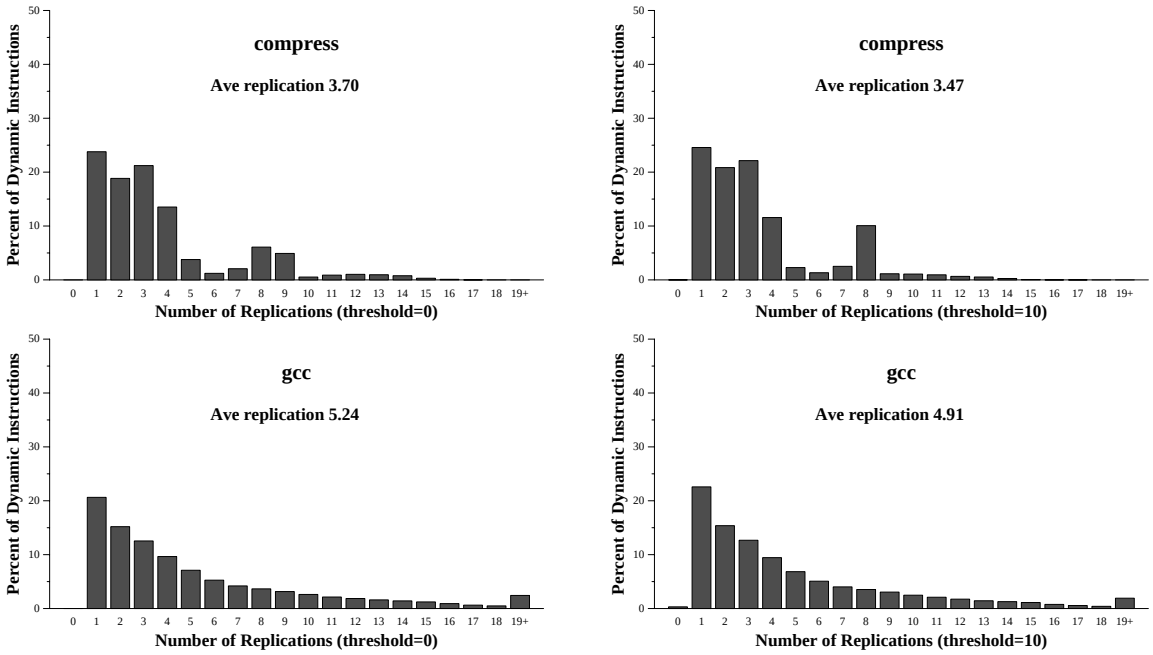

Figure 8.8: Replication distribution for with thresholds of zero and ten, part 1.

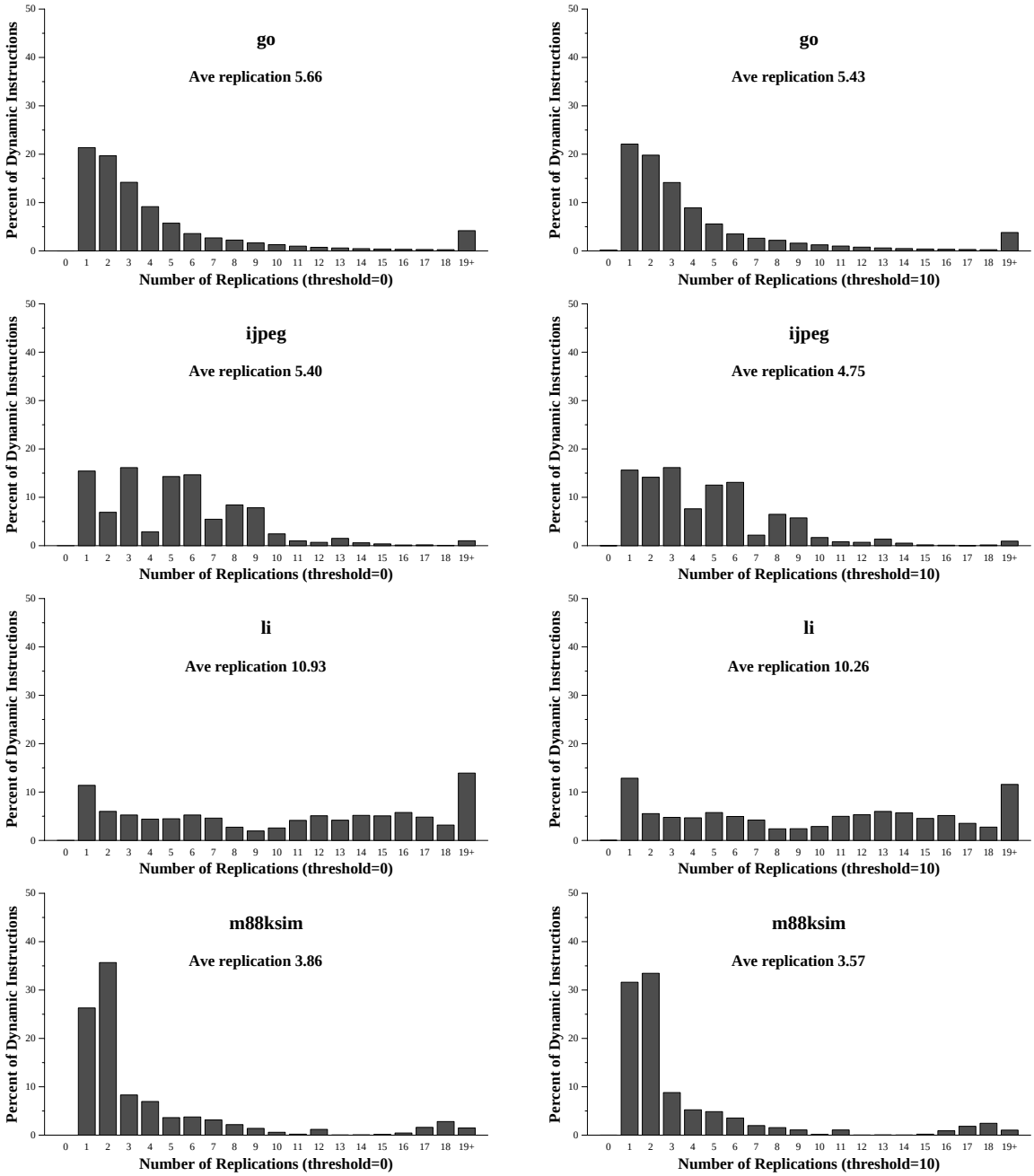

Figure 8.9: Replication distribution for with thresholds of zero and ten, part 2.

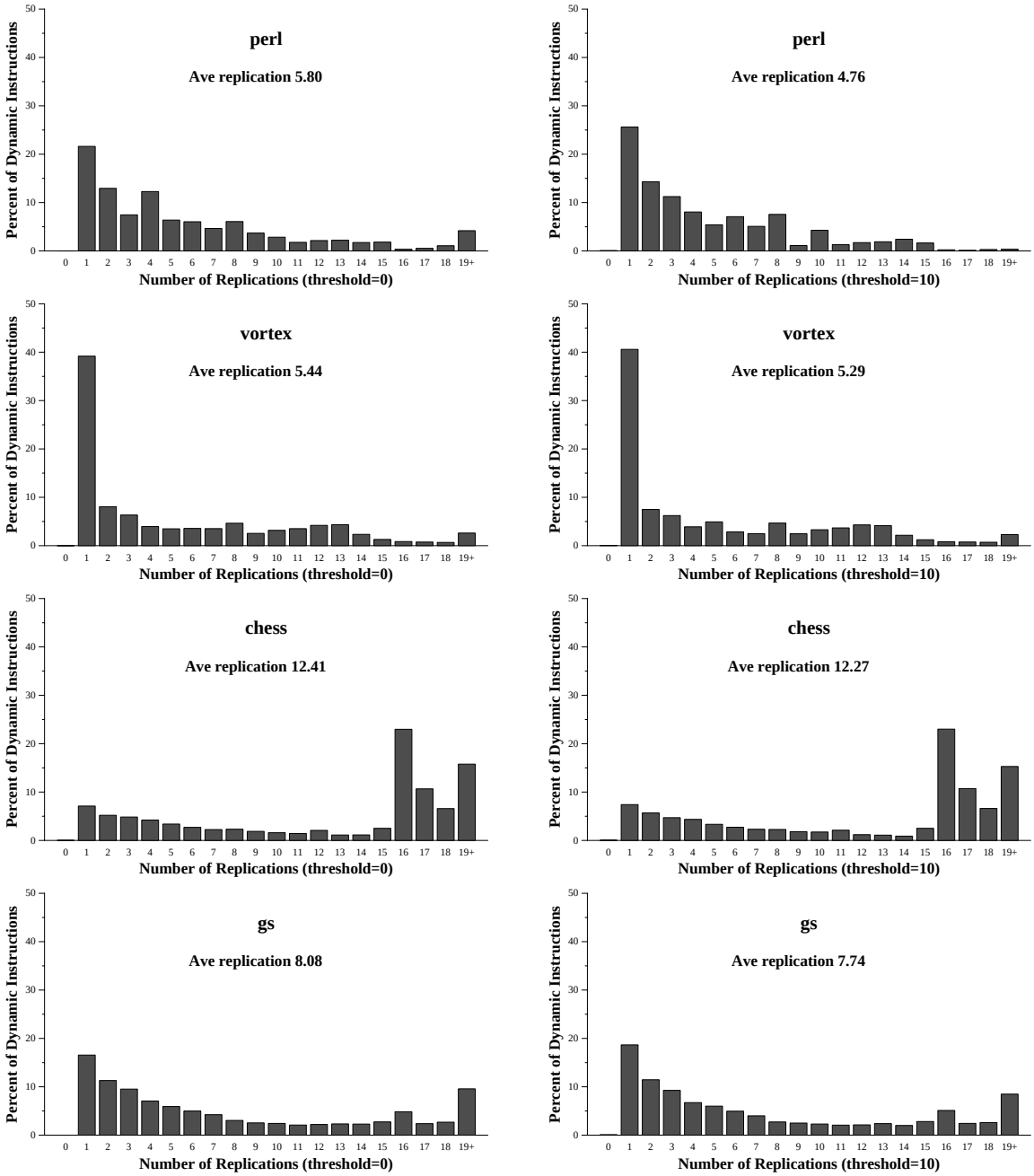

Figure 8.10: Replication distribution for with thresholds of zero and ten, part 3.

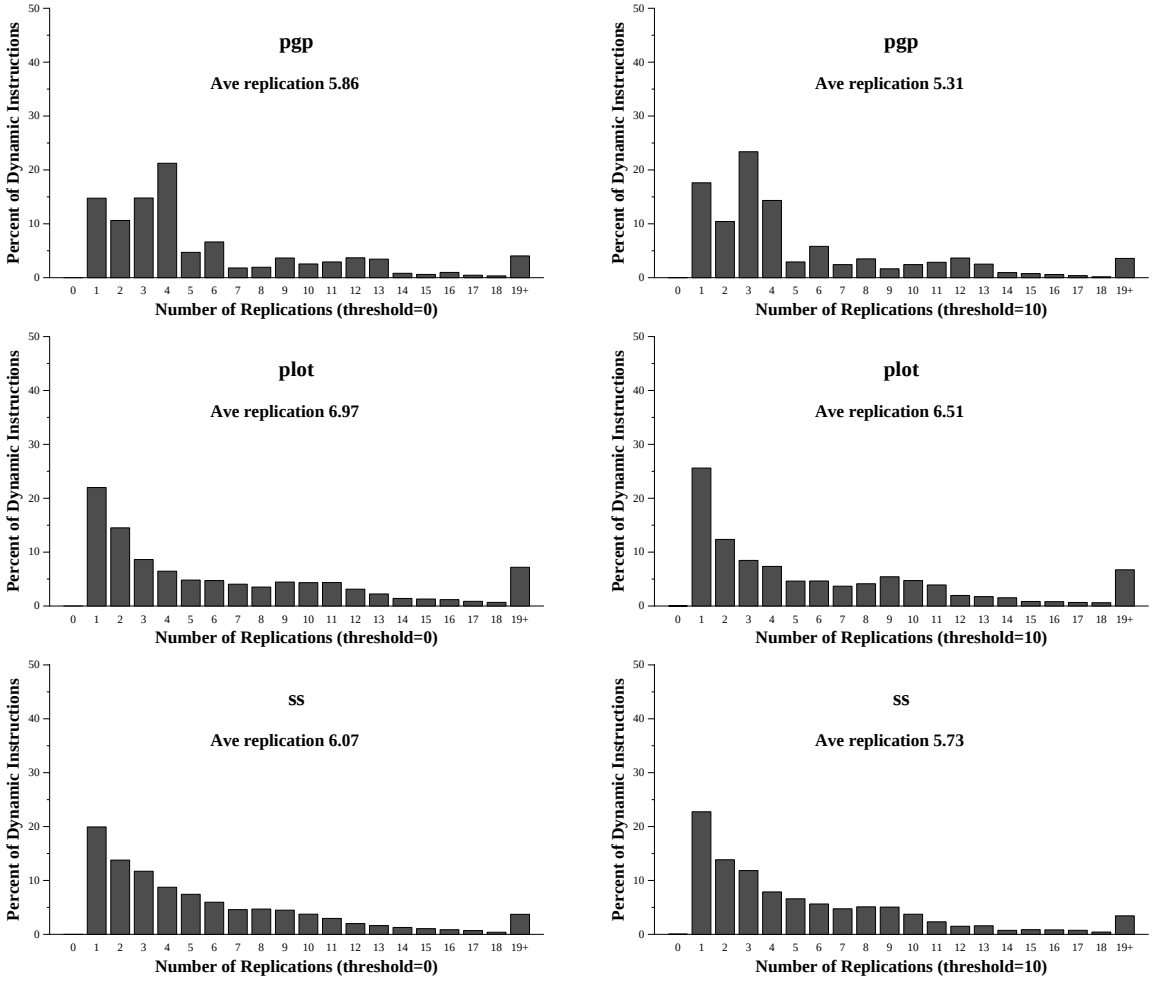

Figure 8.11: Replication distribution for with thresholds of zero and ten, part 4.

To summarize the data contained in the previous figures, the combined average of all the data is presented in following figures. Figure 8.12 shows the average duplication over all the benchmarks, Figure 8.13 shows the average replication, with *threshold*=0, and Figure 8.14 shows the average replication, with *threshold*=10. In the average case, there are 6.81 copies of an average instruction each time it is read from the trace cache. Of those, only 6.57 copies of it will ever be read, and 6.15 copies of it will be read more than 10 times. Table 8.2 presents these numbers per benchmarks in tabular form. The conclusion to be drawn from this is that the trace cache mechanism creates many copies of an instruction, and relies on most of these copies in delivering it's bandwidth.

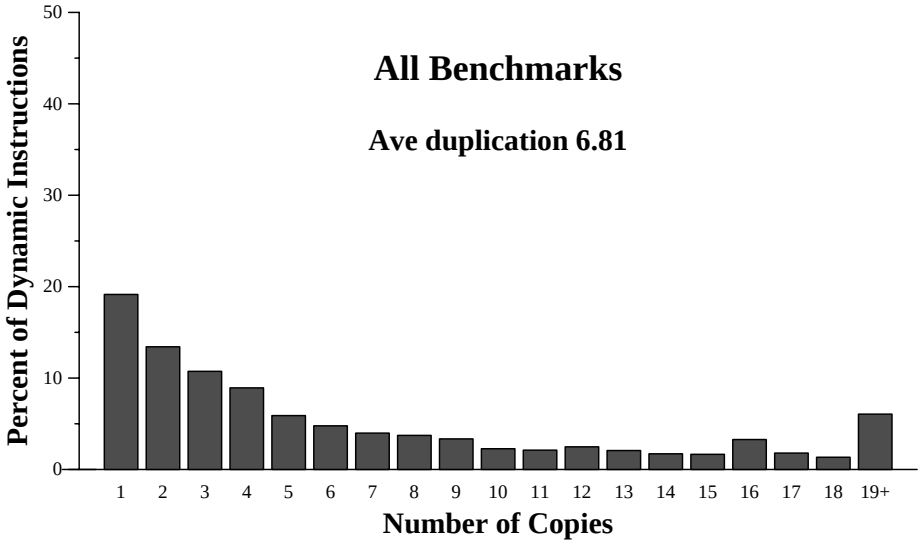

Figure 8.12: The average duplication over all benchmarks.

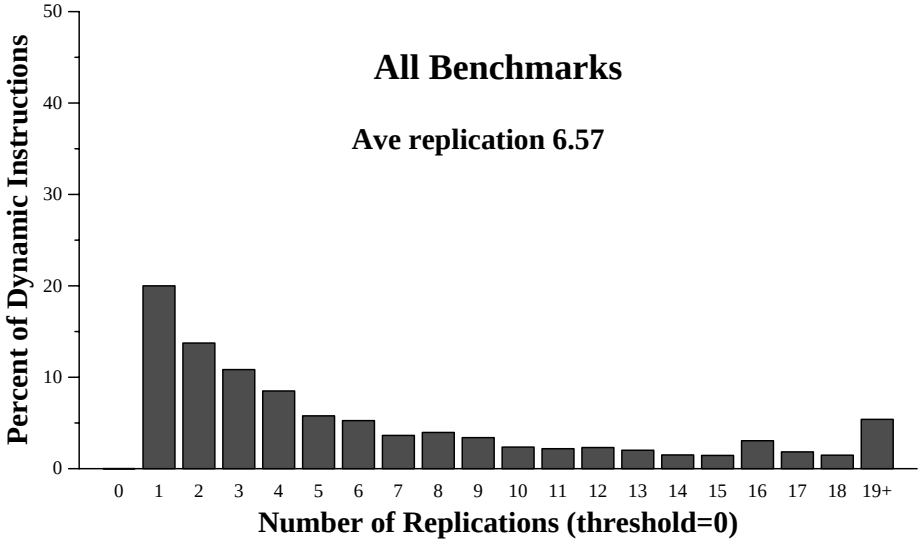

Figure 8.13: The average replication over all benchmarks, threshold = 0

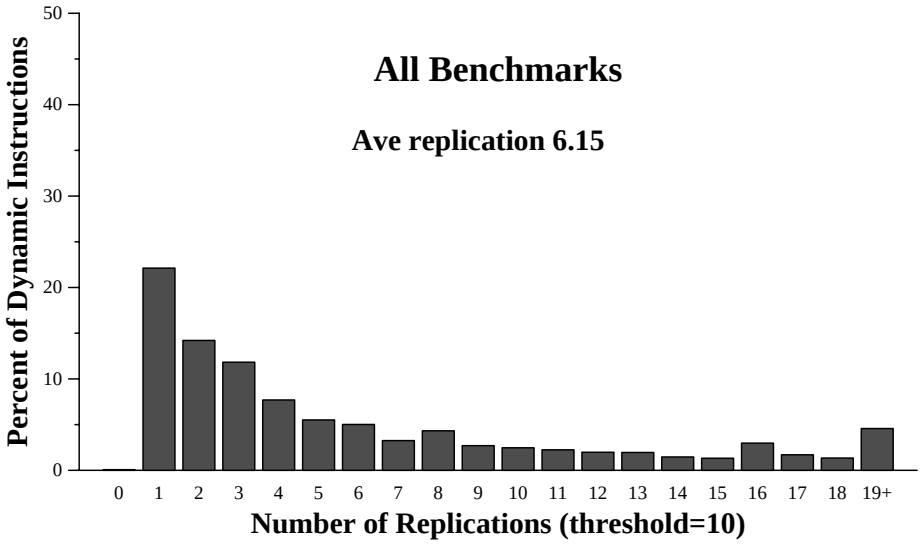

Figure 8.14: The average replication over all benchmarks, threshold = 10.

Benchmark	Duplication	Replication threshold = 0	Replication threshold = 10
compress	4.08	3.70	3.47
gcc	5.37	5.24	4.91
go	5.72	5.66	5.43
ijpeg	5.61	5.40	4.75
li	11.33	10.93	10.26
m88ksim	4.04	3.86	3.57
perl	6.38	5.80	4.76
vortex	5.51	5.44	5.29
chess	12.51	12.41	12.27
gs	8.19	8.08	7.74
pgp	6.43	5.86	5.31
plot	7.23	6.97	6.51
ss	6.17	6.07	5.73
Average	6.81	6.57	6.15

Table 8.2: The average number of copies of an instruction.

8.3.4 Filtering out redundancy

This experiment examines the filtering of redundant trace segments. The objective here is to examine the impact on performance by reducing the amount of redundantly cached instructions. Trace segments which were deemed to be redundant based on the trace segment profile of a previous run are prevented from being written into the trace cache. As established in the previous subsection, trace segments which are accessed fewer

than or equal to a *threshold* number of times in total are considered useless.

Figure 8.15 shows the performance implications of such write filtering of the trace cache at thresholds of zero and ten. With the threshold set to zero, trace segments which are never going to be read are not written into the trace cache. Provided that the initial conditions to the benchmark are the same, such a strategy has a slight beneficial effect.

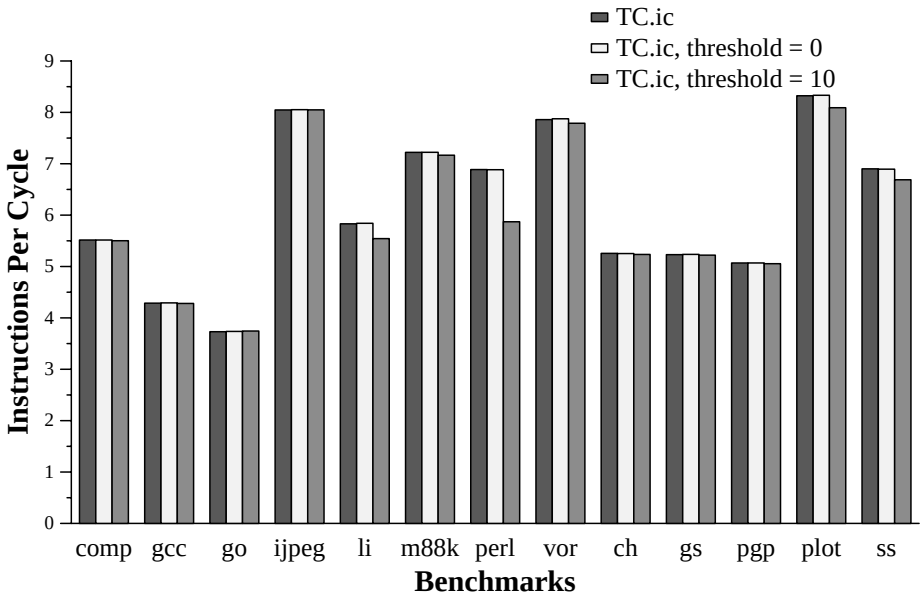

Figure 8.15: The performance of the Enhanced TC.ic with write filtering.

However, with the threshold set to ten, a very interesting phenomenon occurs. There is a slight negative effect in that some trace segments which are read very infrequently are not going to be resident in the trace cache, thus a slight amount of bandwidth will lost. There is also a positive effect: fewer useful segment will be displaced by these useless segments. But for some benchmarks (perl, li), a huge loss in performance is observed. By always filtering the write of these useless segments, the execution conditions are modified in such a way that the filtered segments actually become necessary. For example, say segment ABC is such a trace which is accessed fewer than ten times (but more than zero times). In the initial run, it could have been resident each time it was accessed. In the measurement run, it would have been not resident any time it was accessed, causing its first block to be fetched from the instruction cache. Now, the sequence of fetch addresses between the initial run and measurement run are different and different trace segments will be accessed. This shift causes the filtered segment to be frequently accessed. A similar situation also exists with *threshold*=0, but on a scale too small to be observed in the figure.

One important impact of the elimination of redundancy is the effect on trace cache misses. Since the benchmarks simulated are relatively small compared to the size of the instruction and trace caches, this is not a major factor on performance for the configurations simulated here but is important for configurations with smaller caches or larger round-trip latencies to the second-level instruction cache. Table 8.3 lists the misses per 1000 instructions without filtering and with filtering at *threshold*=0.

Benchmark	without filtering	with filtering threshold = 0
compress	0.00	0.00
gcc	4.04	4.00
go	5.69	5.63
ijpeg	0.00	0.00
li	0.00	0.00
m88ksim	0.04	0.03
perl	0.30	0.25
vortex	1.27	1.23
chess	0.22	0.22
gs	1.38	1.35
pgp	0.01	0.01
plot	0.05	0.05
ss	0.42	0.39
Average	1.77	1.73
Overall i-fetch	0.81	0.79

Table 8.3: Trace cache misses per 1000 instructions with and without trace write filtering.

8.3.5 A strategy to reduce redundancy

Duplication is a result of execution paths through a program which merge at certain blocks. Examples of such blocks include the join blocks of if-then-else conditional structures, blocks at the beginning of loops, and blocks at the beginning of subroutines. In order to eliminate duplication, it would be sufficient to terminate all trace segments at all merge blocks. This would have the effect of synchronizing segment creation of all possible execution paths. But, the cost of such a strategy would be detrimental to performance because such points occur frequently.

In this experiment, a class of merge blocks force trace segment synchronization. Here,

all blocks at the beginning of subroutines start a trace segment, i.e., call instructions force a pending trace segment to finalize. The result is that trace segments are created from a uniform starting point for all subroutines, regardless of caller of the subroutine. Figure 8.16 shows the average duplication for three variations of the TC.ic configuration. The first two are the same as Figure 8.5—the first is without trace packing, the second is with cost-regulated trace packing. The final scheme finalizes trace segments on call instructions. For the final scheme, average duplication is significantly reduced. Figure 8.17 demonstrates the performance impact of such a strategy. For some benchmarks, this reduction in duplication comes at a minor cost.

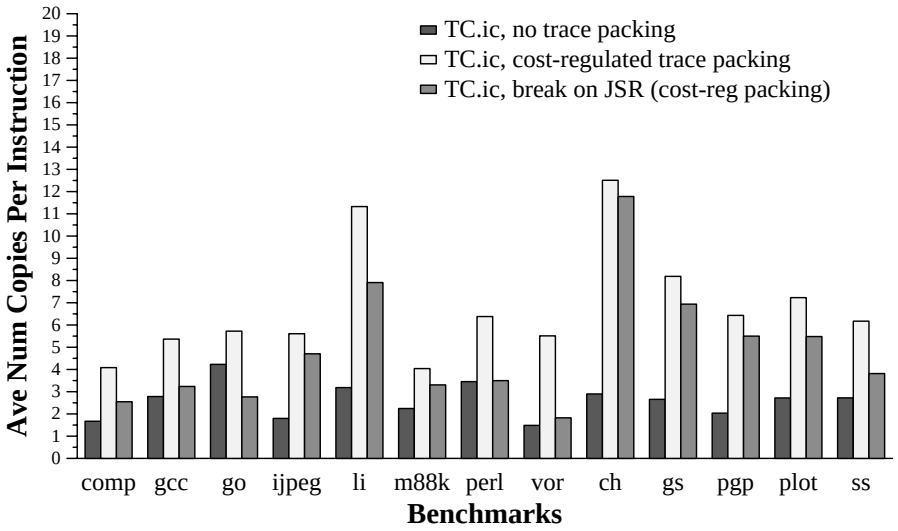

Figure 8.16: The effect of finalizing on call instructions on average duplication.

8.3.6 Unused space

Similar to duplication, another effect that degrades the caching efficiency of the trace cache is the unused cache space due to incomplete trace cache lines. Since the trace segment construction rules finalize segments even if it is not full, e.g., when the third branch is added, many trace segment often contain fewer than 16 instructions. The effect of this unused space is higher miss rates and lower effective fetch rates. It should be noted that instruction caches suffer from a similar problem: because long cache lines can contain several sequential basic blocks, a cache line may contain a block of instructions that never gets executed but was cached because of a fetch to another block on that same line.

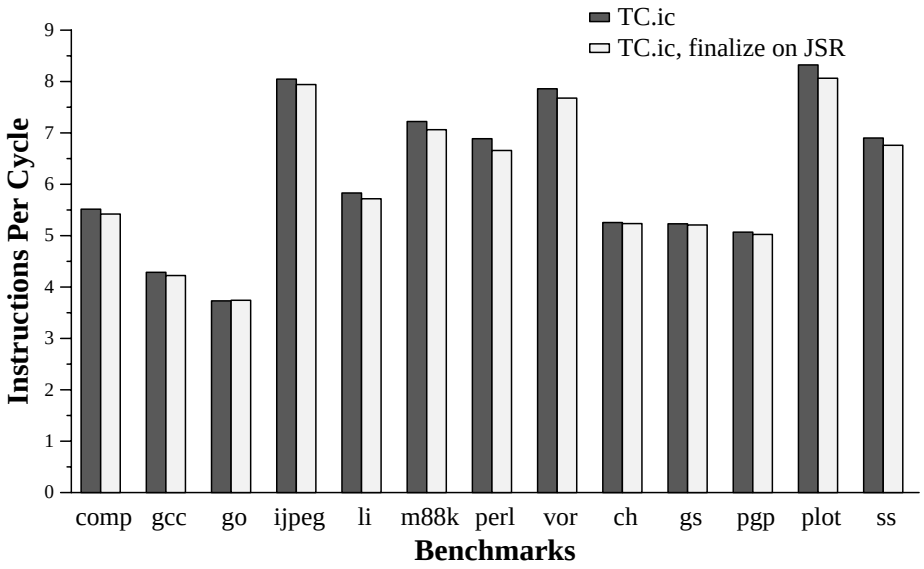

Figure 8.17: The performance implications of finalizing on call instructions.

To gauge the amount of the unused space due to incompletely filled trace cache lines, the unused instruction slots on each line fetched from the trace cache are counted and averaged over all trace cache fetches. These averages are presented in Table 8.4. The data are gathered on two TC.ic configurations: no trace packing and cost-regulated trace packing.

Benchmark	No Trace Packing	Cost-reg Packing
compress	3.48	2.03
gcc	4.18	3.29
go	3.93	3.20
ijpeg	2.49	1.26
li	3.49	2.20
m88ksim	3.82	2.64
perl	3.77	3.08
vortex	4.45	2.63
chess	2.91	1.11
gs	3.93	2.51
pgp	3.35	1.67
plot	3.76	2.27
ss	3.82	2.34
Average	3.64	2.33

Table 8.4: The number of unused trace cache instruction slots per fetch.

8.4 Large Atomic Units

An execution atomic unit (EAU) [32] is a sequence of instructions in which all instructions execute or none do. A typical execution atomic unit can be sequence of instructions delimited by a conditional branch or an by indirect branch or by a return instruction. However, a particular processor implementation may divide (or enlarge) that sequence based on, for example, fetch width, issue width or execution resources.

Large EAUs are beneficial to the underlying hardware because they allow resource allocation on a larger unit of work. For example, an EAU only requires a single checkpoint. Another example is that architectural registers need only be read and written on values which cross EAU boundaries; internal communication can be explicitly encoded within the EAU.

From the perspective of the instruction fetch mechanism, EAUs represent a block of instructions fetched at the cost of one control decision. Since conditional and indirect branches, and returns terminate an EAU, fetching a single EAU requires a single branch (or target) prediction in order to determine the next EAU to be fetched. From the perspective of the fetch mechanism, large EAUs represent more instructions fetched for each prediction made, implying that less effective bandwidth is required from the branch prediction mechanism.

The trace cache, in general, and Branch Promotion, specifically, enable EAUs to be enlarged by removing artificial terminations. The trace cache allows trace segments formation beyond unconditional branches and subroutine calls. Branch Promotion allows for highly biased conditional branches to be treated like unconditional branches. With these types of control instructions removed from consideration, EAU size now is terminated by only non-promoted conditional branches, indirect branches, and return instructions. The effects of this are shown in Figure 8.18 which shows EAU size distribution averaged over all the benchmarks, executed on the Enhanced TC.ic configuration.

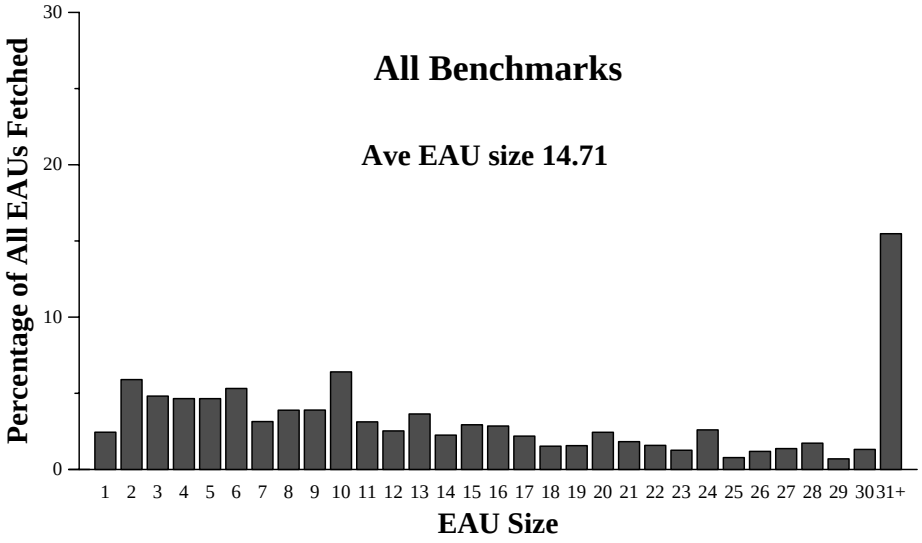

Figure 8.18: The distribution of EAU sizes during execution on the Enhanced TC.ic configuration, averaged over all benchmarks.

CHAPTER 9

Trace Cache Design for Next Generation Processors

9.1 Objective: a Fetch Mechanism for an 8-wide Machine

Section 8.4 demonstrates that the trace cache can transfer, on average, a large number of instructions for each control transfer instruction. In this chapter, the trace cache, based on this notion, is used to develop a high-bandwidth instruction delivery mechanism for a next-generation 8-wide issue processor.

The fetch mechanism proposed for the next-generation consists of a 64KB trace cache with Branch Promotion. The trace segments of this mechanism consist of 8 instructions with at most 1 conditional branch or indirect jump or return. Since promoted branches are not considered conditional branches, there is no limit on their number (except the implicit maximum of 8). This design allows for two things. First, an aggressive single branch predictor can be used, such as the hybrid predictor used in the Single-Block ICache configuration. Second, the design allows for a high frequency implementation. The selection of next fetch address is simpler because at most one branch is fetched. This fetch mechanism gets high instruction bandwidth via both a high fetch rate and a low branch misprediction rate and is more nearly amenable to short cycle times.

In this chapter, this next-generation trace cache mechanism is described and compared with a comparable Single-Block ICache and the Sequential-Block ICache (with multiple branch predictor).

9.2 Trace Cache with Promotion, 1 Branch Only

The fetch mechanism consists of a 64KB trace cache and a supporting 4KB icache. The trace cache consists of lines of 8 instructions, each line containing up to one conditional branch or indirect jump or return instruction. The fill unit performs Branch Promotion, and a line may contain multiple promoted branches. The threshold for Promotion is 64. Since the trace cache only stores one (extended) block per trace segment, the other enhancements, Partial Matching, Inactive Issue, and Trace Packing, do not apply.

9.3 Aggressive Single Branch Predictor

Since only a single branch prediction per cycle is required with this fetch mechanism, a larger range of predictor options are available. One of the best performing predictors documented in the branch prediction literature is a gshare/PAs hybrid [6, 12, 29] similar to the one used on the Digital/Compaq 21264 [27] (See Section 4.6.1 for a description). This type of predictor is able to predict both branches which exhibit global branch correlation and branches which correlate best to themselves [13]. In the configuration examined here, the global component consists of an 8KB pattern history table, the per-address component a consisting of an 8KB pattern history table and a set of 2K 15-bit branch histories (stored in a Branch Target Buffer), and an 8KB selector mechanism.

Because of the trace cache's dynamic nature, branch prediction with this mechanism performs better if it is based on *branch* address rather than fetch address. With a trace cache, a branch can be accessed using a large number of fetch addresses, due to the fact that trace segments grow and shrink due to Branch Promotion and other factors which vary trace segment construction. This increase in the mappings between fetch addresses and branches decreases the efficiency of the branch predictor, since information on the same branch is now spread across multiple entries. To circumvent this problem, alongside each next fetch address is stored a next branch ID (it need not be a full copy of the branch's address). The fetch address is used to access the trace cache and instruction cache, whereas the branch ID is used to access the branch prediction structures. In the case that the branch ID is not correct (which can happen if branch ID is a subset of the address or if promotion has changed the fetch-address-to-branch mapping), then branch ID is repaired.

9.4 8-wide Execution Engine

This fetch mechanism is coupled to an 8-wide execution engine through a window of 256 instructions. The window is divided into 8 columns of 16 instructions, each column feeding a multi-purpose execution unit. The machine contains a 32KB single-cycle data cache with 5 read/write ports. All other parameters of the execution engine are similar to those described in Section 4.4.

9.5 Comparison to Alternatives

Figure 9.1 demonstrates the performance advantage in Instructions Per Cycle of the 8-wide trace cache mechanism over the Single-Block ICache and the Sequential-Block ICache. The Sequential-Block ICache delivers up until the first taken branch, up to three branches total. For these comparisons, both icache configurations were scaled back to 64KB. The percentages above the bars indicate the performance differential between the Sequential-Block ICache and the trace cache. On most benchmarks, the trace cache outperformed the icache configurations. On gcc, the Sequential-Block ICache outperforms the trace cache.

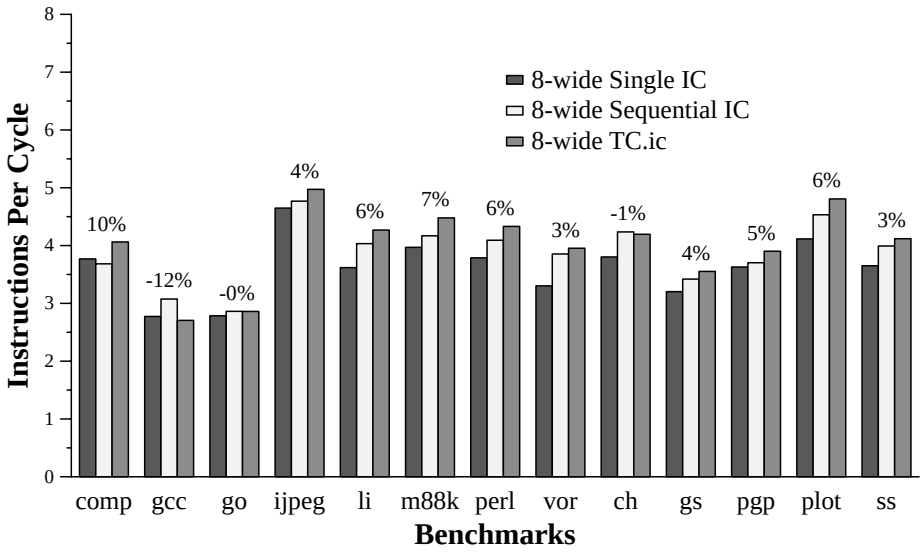

Figure 9.1: The performance of the three 8-wide fetch mechanisms.

The trace cache gets its primary performance advantage over the Sequential-Block ICache via highly accurate branch prediction. Table 9.1 lists the branch misprediction rates of the three configurations tested.

Furthermore, the trace cache delivers a high effective fetch rate. Figure 9.2 shows the

Benchmark	Single-Block ICache	Sequential-Block ICache	Trace Cache
compress	5.34	8.88	5.40
gcc	6.48	8.15	7.11
go	15.26	17.71	15.34
ijpeg	8.53	9.48	8.42
li	2.84	4.20	3.73
m88ksim	0.86	1.96	0.96
perl	1.26	3.06	1.44
vortex	0.95	1.85	1.11
chess	1.91	2.45	2.04
gs	4.27	5.56	4.33
pgp	4.24	5.49	4.19
plot	1.53	2.61	1.66
ss	3.49	5.31	3.68
Average	4.38	5.90	4.57

Table 9.1: The branch misprediction rates of the three 8-wide fetch mechanisms.

effective fetch rate in instructions per fetch. The trace cache configuration delivers a slightly higher fetch rate for most benchmarks than the Sequential-Block scheme. But because of the low frequency of promoted branches and the prevalence of small basic blocks, the benchmarks gcc and go fare poorly in terms of fetch rate with this trace cache configuration.

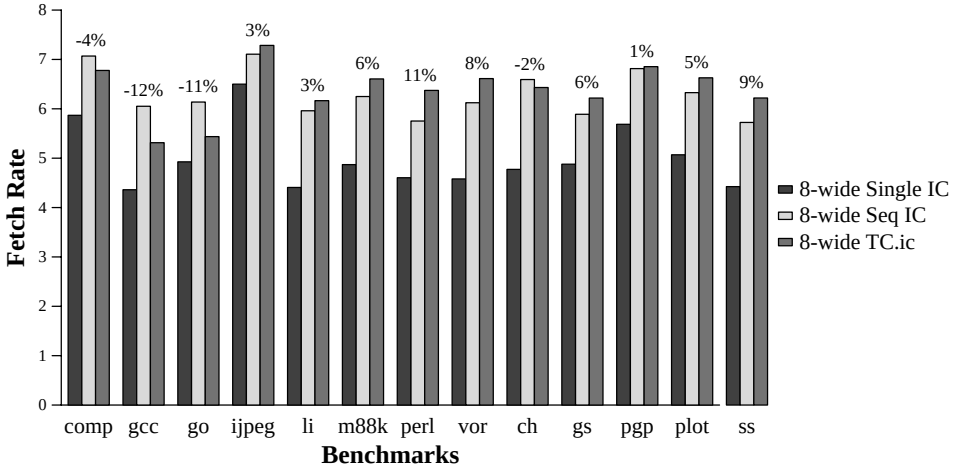

Figure 9.2: The effective fetch rate of the three 8-wide fetch mechanisms.

Finally, the trace cache configuration suffers from a larger number of fetch misses, due primarily to the duplication effect discussed in Section 8.3. Table 9.2 lists the overall

fetch misses (i.e., fetch requests which miss both caches in the case of the trace cache configuration) per 1000 instructions.

Benchmark	Single-Block ICache	Sequential-Block ICache	Trace Cache
compress	0.00	0.00	0.01
gcc	2.54	2.59	17.29
go	3.82	3.22	10.60
ijpeg	0.00	0.00	0.00
li	0.00	0.00	0.10
m88ksim	0.01	0.03	0.97
perl	0.06	0.18	5.19
vortex	2.25	2.25	11.37
chess	0.03	0.03	2.84
ghostscript	1.93	1.93	4.38
pgp	0.01	0.01	0.03
plot	0.03	0.03	0.97
sim-outorder	0.12	0.17	8.46
Average	0.71	0.69	3.73

Table 9.2: The fetch misses per 1000 instructions for the three 8-wide fetch mechanisms.

CHAPTER 10

Conclusion

The trace cache is a mechanism for overcoming the partial fetch problem by caching logically contiguous instruction in physically contiguous storage. By caching instruction in this dynamic manner, the trace cache is able to deliver instruction at a high effective fetch rate—effective fetch rate is the number of good instructions fetched for each fetch that hits in the caches. The primary objective of this dissertation is that of improving the effective fetch rate of the trace cache mechanism.

The effectiveness of the trace cache is improved by a series of enhancements proposed in this dissertation. As a result, the trace cache is able to outperform a Sequential-Block ICache. This ICache is capable of fetching sequential runs of instructions and utilizes a compiler optimization which increases run lengths. The enhanced trace cache outperforms this ICache by 14% in IPC and by 34% in average fetch rate. The enhanced trace cache outperforms a trace cache without these enhancements by 22% in IPC and by 35% in fetch rate. In order for the trace cache to outperform the Sequential-Block ICache, the enhancements are necessary.

The first enhancement evaluated is Partial Matching and it enables the trace cache to deliver a portion of a cached trace segment, based on the path selected by the branch predictor. This policy significantly increases the number of fetches serviced by the trace cache.

Inactive Issue is a further refinement of Partial Matching in that blocks in a trace segment not selected by the predictor are issued *inactively* as a hedge against a branch misprediction. If the adjoining branch is later detected to be mispredicted, then the processor can recover slightly further along the correct path because of the inactively issued instruc-

tions. Inactive Issue boosts the performance of benchmarks which suffer a high degree of branch mispredictions.

Of the factors which limit trace size, the maximum branch limit of three conditional branches is most significant. Branch Promotion was designed to address this limitation. It does so by converting highly biased branches into unconditional branches which have the capability to generate a fault if they change direction. Promoted branches do not require a dynamic prediction and thus do not affect the three branch limit per trace segment. With branch promotion, an average of 60% of dynamic conditional branches are removed from the prediction stream. The result is lower interference in the predictor, less reliance on multiple branch predictions per cycle, and more instructions per trace segment.

Branch Promotion increases fetch rate only slightly. The default trace construction strategy used by the fill unit is to not split blocks of instructions across trace segments. Branch Promotion results in more fetches limited by this fill unit policy. Trace Packing is a scheme whereby the fill unit can split blocks arbitrarily. With both Branch Promotion and Trace Packing, trace cache fetch rate is boosted by 17%. Both enhancements are synergistic in that each addresses the limitation suffered by the other. The main drawback of Trace Packing is that it significantly increases the number of copies of an instruction cached in the trace cache. A Trace Packing strategy which only packs segments if the potential payoff is high is proposed and demonstrated to give high fetch rates with low cache miss rates.

Of the three trace cache configurations tested, the TC.ic scheme performs best. It is slightly better than the scheme where storage is split between the two. As the effectiveness of the trace cache is further improved with additional enhancements (such as a trace cache where trace segments are optimized for fast execution), the relative performance of this scheme is expected to increase.

The trace cache outperforms the alternative configurations and allows for logic complexity to be moved out of the critical processing path of the machine. As demonstrated, fill unit latency does not affect processor performance. Some operations which typically are placed between the fetch stage and the issue stage can be moved into the fill unit pipeline.

Even though performance of the entire processor increases by a modest 14% over the Sequential-Block ICache, the real problem addressed by the trace cache is that of fetch rate. The fetch rate is demonstrated to increase by 34% with the enhanced trace cache. Other factors, such as execution rate, limit the performance attainable with the machine modeled

in this dissertation.

One of these other factors is branch resolution time. As fetch rate is increased, branch resolution time increases causing more cycles to be lost due to branch mispredictions. This increase is due to the accumulation of processing delays associated with the execution hardware, such as resource contention, cache misses, etc. In addition, instruction level parallelism plays a slight role in that as a branch is fetched earlier, it must wait longer for the natural dataflow to progress before being able to execute.

The caching of instructions in logical order opens the possibility of caching duplicate copies of instructions. The measurements presented in Section 8.3 demonstrate that Trace Packing increases the amount of duplication from an average of three copies of an instruction to about eleven. Cost-regulated Trace Packing moderates duplication to about seven copies per instruction. This duplication can be divided into useful replication, where a copy is necessary for delivering high bandwidth, and useless redundancy, where a copy only occupies cache space. Measurements indicate that about six of the seven cached copies of the average instruction contribute to the fetch rate.

Finally, an implementation for next-generation processor microarchitectures is described. The scheme outlined in this dissertation gets high fetch rates without compromising high branch prediction accuracy. For most benchmarks, this 8-wide fetch scheme attains higher performance than the 3-branch Sequential-Block ICache.

APPENDICES

APPENDIX A

Lessons Learned from the Trace Cache

Several techniques which boosted the trace cache's effectiveness were described in in Chapter 6. Two of these techniques can be applied to the icache configurations as well. In this appendix, Inactive Issue and Branch Promotion are adapted for use with the Sequential-Block ICache.

A.1 Applying Inactive Issue to the Sequential ICache

The technique of Inactive Issue is not restricted to the trace cache. The concept can be applied to any multi-block fetch mechanism. Blocks fetched but not selected by the branch predictor can be issued inactively. In this experiment, the technique is applied to the Sequential-Block ICache.

Here, the Sequential-Block ICache fetches up to three sequential fetch blocks. While the icache is being accessed, the branch predictor is queried for three predictions. Based on the predictions, the fetched blocks are issued actively or inactively. The active portion of the fetch terminates with the first predicted taken branch. The subsequent blocks are issued inactively.

Figure A.1 demonstrates the performance implications of adding Inactive Issue to the standard Sequential-Block ICache. The numbers are not qualitatively different than with the trace cache. Many benchmarks exhibit little or no change in performance. However, benchmarks which suffer more significantly from branch mispredictions, such as the benchmarks gcc and go, benefit from Inactive Issue. Some benchmarks suffer slightly (compress, ijpeg, pgp) because inactive instructions are also executed and thus compete with active

instructions for execution bandwidth.

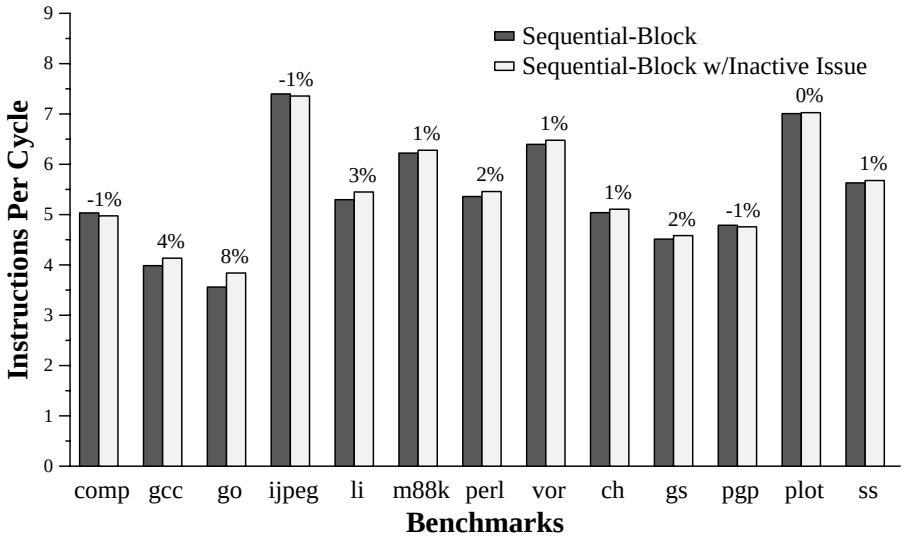

Figure A.1: The Sequential-Block ICache with and without Inactive Issue.

A.2 Applying Branch Promotion to the Sequential-Block ICache

Branch Promotion can also be applied to the Sequential-Block ICache. By doing so, the number of fetches limited by the 3 (not-taken) branch limit can be reduced. To implement Branch Promotion with the icache, a mechanism must exist to mark and store branch instructions which satisfy the promotion criterion as promoted. Because of the nature of the Sequential-Block ICache, only not-taken promoted branches are helpful in boosting fetch rate.

A.2.1 ICache fetch terminations

Figure A.2 shows a fetch breakdown for the Sequential-Block ICache running the benchmark gcc. Here, there are five reasons which can limit a fetch:

1. Taken BR. The fetch was terminated by a taken branch (conditional, unconditional, or jsr) or a cache line boundary [1] before 16 instructions were fetched.
2. Mispred BR. A mispredicted branch terminated the fetch.
3. Max Size. The fetch contained 16 instructions.
4. Ret, Indir, Trap. A return, indirect jump, or trap terminated the fetch.
5. Maximum BRs. The current fetch contained three not-taken conditional branches.

The number of fetches terminated by the maximum branch limit is sizable, but it is not the only large factor causing partial fetch terminations. Many small fetches result because of the taken branch limitation.

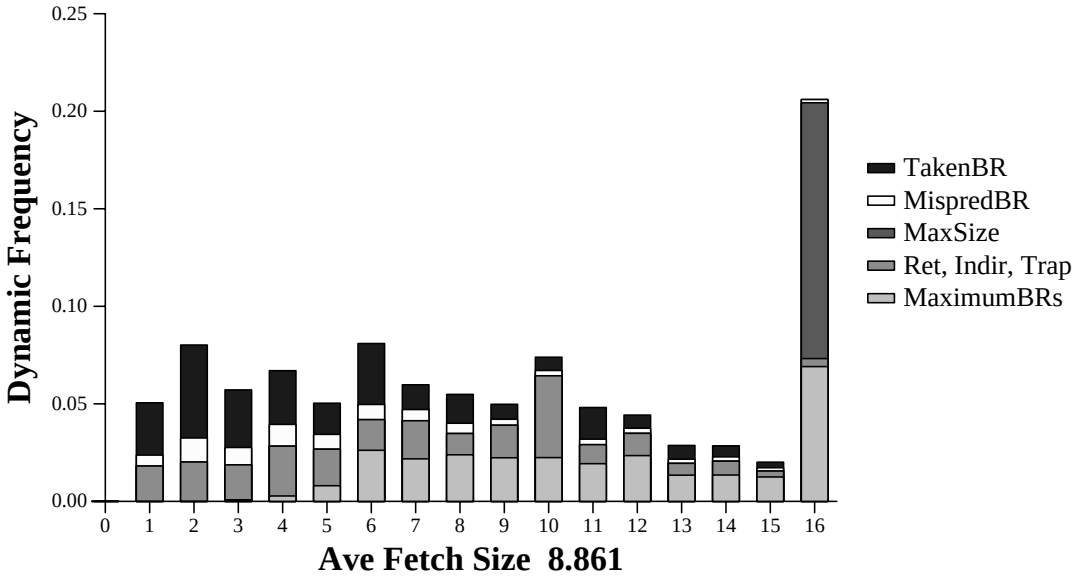

Figure A.2: Fetch termination reasons for the Sequential-Block ICache on gcc.

A.2.2 Applying promotion

The concept of promotion is described in Section 6.6. With the Sequential-Block ICache, conditional branches which are almost always fall-through are signaled for promotion and

[1] The instruction cache implements split-line fetching, however cache line boundaries can still terminate a fetch if the request for the second line misses in the cache.

are modified in the icache as promoted branches. Promoted branches do not require a branch prediction when they are fetched, instead they are statically predicted to be not taken. If the branch behaves differently, a hardware fault is signaled and the branch is corrected. With this mechanism, fewer fetches are limited by the three branch limit.

Figure A.3 shows the performance implications of performing Branch Promotion on the Sequential-Block ICache configuration. The threshold for promotion is set to 64 not-taken occurrences. All benchmarks get a very slight performance boost from Promotion.

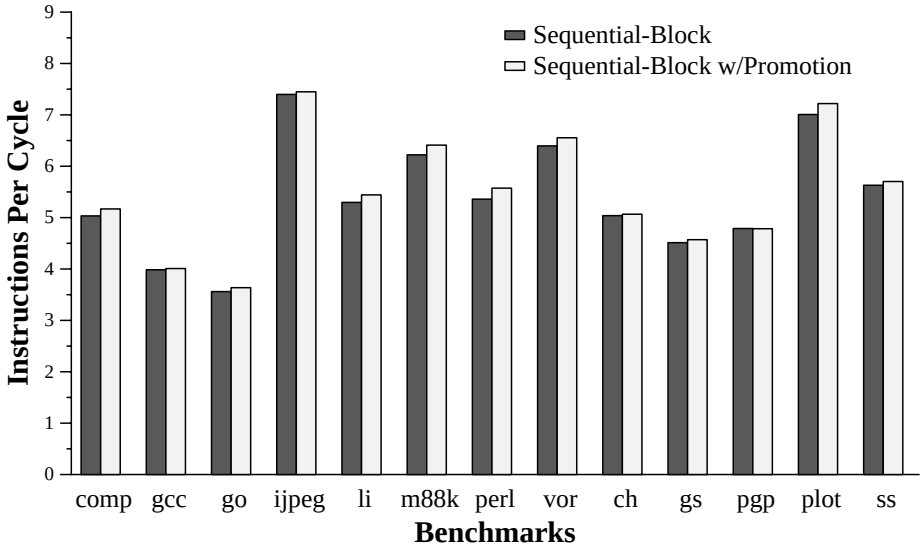

Figure A.3: The Sequential-Block ICache with and without Branch Promotion.

The boost is not significant because promotion is not addressing the largest factor causing small fetches. Figure A.4 shows a fetch termination histogram for the Sequential-Block ICache with Branch Promotion activated. Here, fewer fetches are limited by the three branch limit as compared to Figure A.2, but a large fraction of smaller fetches are due to the taken branch limitation. The essential problem with the Sequential-Block ICache is that the large sequential runs of instructions created by the compiler are fragmented by the fetch hardware. For instance, a sequential run of 17 instructions is fragmented into a fetch of 16 instructions and a subsequent fetch of 1 instruction, resulting in an overall fetch rate of 8.5.

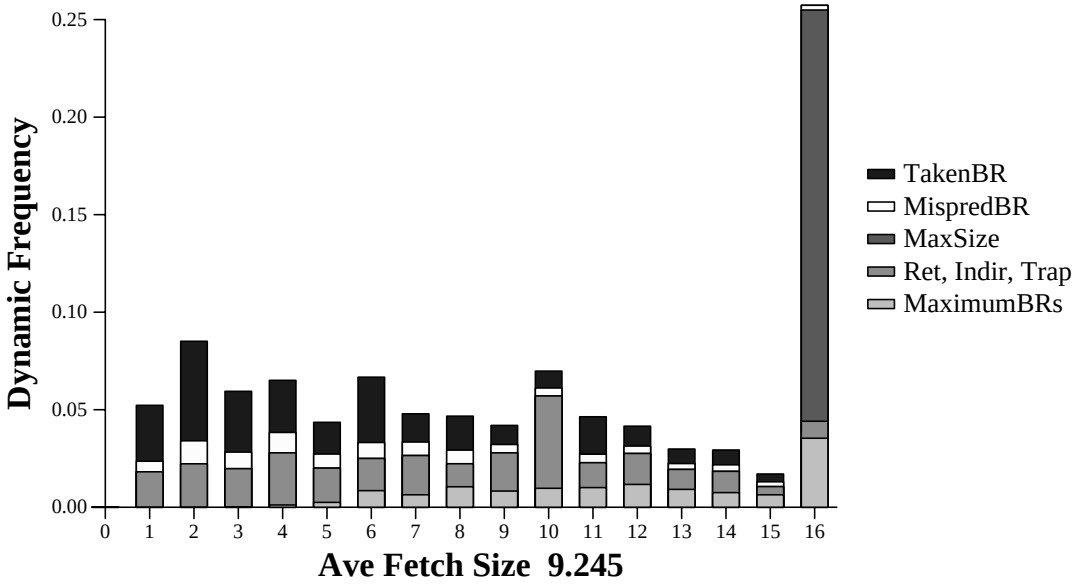

Figure A.4: Fetch termination reasons for the Sequential-Block ICache with Promotion on gcc.

A.3 Enhanced Sequential-Block ICache

The Enhanced Sequential-Block ICache is created by adding both Inactive Issue and Branch Promotion to the basic icache. Figure A.5 demonstrates performance of the Sequential-Block ICache, the Enhanced Sequential-Block ICache, and the Enhanced Trace Cache (TC.ic). Overall, the Trace Cache performs 12% better than the Enhanced Sequential-Block ICache and 14% better than the icache without the enhancements. The numbers above the bars indicate the performance improvement of the Trace Cache over the Enhanced ICache.

The enhancements are all about boosting the fetch rate. Figure A.6 demonstrates the boost in fetch rate by adding these enhancements. Overall the trace cache has a 28% larger fetch rate than the Enhanced ICache. The Enhanced ICache has a 5% larger fetch rate than the ICache without the Enhancements.

The Enhanced ICache outperforms the Trace Cache on the benchmark go (See Figure A.5) even though the Trace Cache gets a higher fetch rate on it. The benchmark suffers a considerable number of cache misses with the Trace Cache–5.69 trace and icache misses per 1000 instructions versus only 0.70 icache misses per 1000 instructions with the Enhanced ICache. The boost in fetch rate with the enhancements is enough to boost the performance of the Enhanced ICache beyond the Trace Cache on this benchmark.

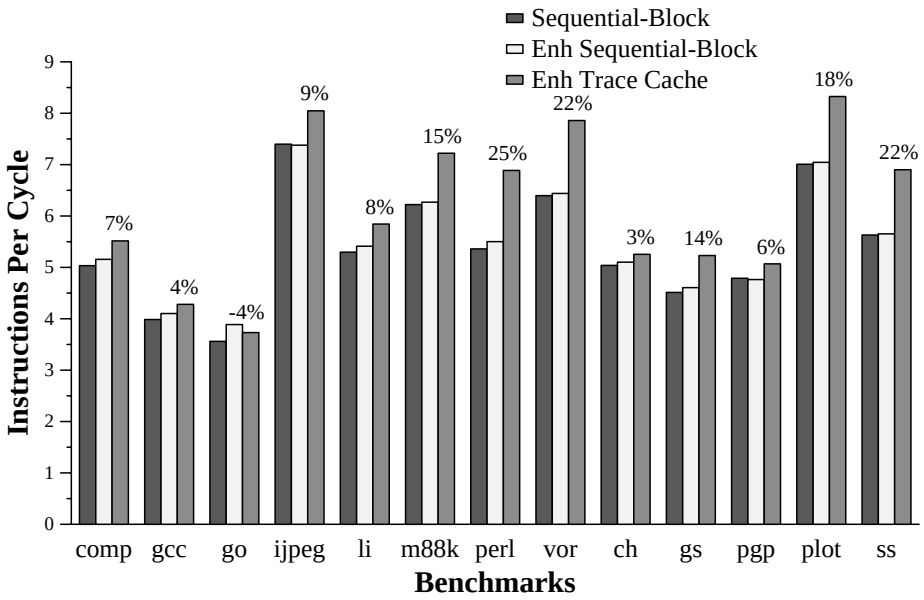

Figure A.5: The performance of the Sequential-Block ICache with both Inactive Issue and Branch Promotion.

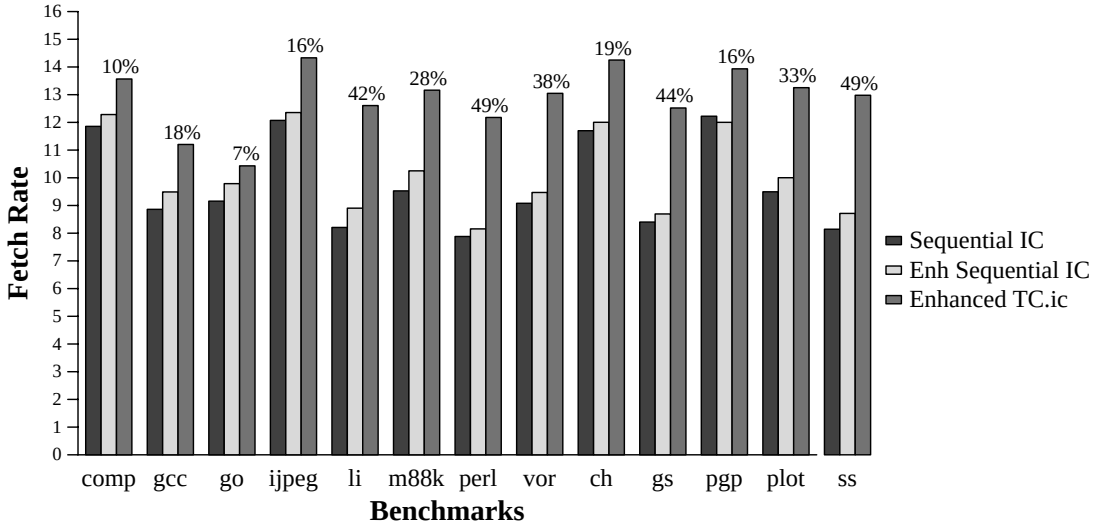

Figure A.6: The fetch rate of the Sequential-Block ICache with both Inactive Issue and Branch Promotion.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] T. M. Austin and G. S. Sohi, "Dynamic dependency analysis of ordinary programs," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 342–351, 1992.

[2] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar tool set," Technical Report 1308, University of Wisconsin - Madison Technical Report, July 1996.

[3] M. Butler and Y. Patt, "An area-efficient register alias table for implementing HPS," in *Proceedings of the 1990 International Conference on Parallel Processing*, pp. 611–612, 1990.

[4] B. Calder and D. Grunwald, "Reducing branch costs via branch alignment," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 242–251, 1994.

[5] P.-Y. Chang, M. Evers, and Y. N. Patt, "Improving branch prediction accuracy by reducing pattern history table interference," in *Proceedings of the 1996 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1996.

[6] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt, "Branch classification: A new mechanism for improving branch predictor performance," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 22–31, 1994.

[7] P. P. Chang and W. W. Hwu, "Trace selection for compiling large c application programs to microcode," in *Proceedings of the 21st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 21–29, 1988.

[8] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 266–275, 1991.

[9] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 142 – 153, 1998.

[10] R. Cohn. Compaq Computer Corporation, February 1999. Personal communication.

[11] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[12] M. Evers, P.-Y. Chang, and Y. N. Patt, "Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 3 – 11, 1996.

[13] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt, "An analysis of correlation and predictability: What makes two-level branch predictors work," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 52 – 61, 1998.

[14] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, July 1981.

[15] M. Franklin and M. Smotherman, "A fill-unit approach to multiple instruction issue," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 162–171, 1994.

[16] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Alternative fetch and issue techniques from the trace cache fetch mechanism," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[17] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors," in *Proceedings of the 31th Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[18] G. F. Grohoski, "Machine organization the IBM RISC system/6000 processor," *IBM Journal of Research and Development*, vol. 34, pp. 72–84, 1990.

[19] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu, "Superblock formation using static program analysis," in *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 247–255, 1993.

[20] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt, "Increasing the instruction fetch rate via block-structured instruction set architectures," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.

[21] P. Hsu and E. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 1986.

[22] W. W. Hwu and Y. N. Patt, "Checkpoint repair for out-of-order execution machines," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 18–26, 1987.

[23] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning confidence to conditional branch predictions," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 142–152, 1996.

[24] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based next trace prediction," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[25] Q. Jacobson and J. E. Smith, "Instruction pre-processing in trace processors," in *Proceedings of the Fifth IEEE International Symposium on High Performance Computer Architecture*, 1999.

[26] J. D. Johnson, "Expansion caches for superscalar microprocessors," Technical Report CSL-TR-94-630, Stanford University, Palo Alto CA, June 1994.

[27] J. Keller, *The 21264: A Superscalar Alpha Processor with Out-of-Order Execution*, Digital Equipment Corporation, Hudson, MA, October 1996. Microprocessor Forum presentation.

[28] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 45–54, 1992.

[29] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[30] S. Melvin and Y. Patt, "Enhancing instruction scheduling with a block-structured ISA," *International Journal of Parallel Programming*, vol. 23, no. 3, pp. 221–243, June 1995.

[31] S. W. Melvin and Y. N. Patt, "Performance benefits of large execution atomic units in dynamically scheduled machines," in *Proceedings of Supercomputing '89*, pp. 427–432, 1989.

[32] S. W. Melvin, M. C. Shebanow, and Y. N. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proceedings of the 21st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 60–63, 1988.

[33] K. N. Menezes, S. W. Sathaye, and T. M. Conte, "Path prediction for high issue-rate processors," in *Proceedings of the 1997 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1997.

[34] R. Nair and 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, 1997.

[35] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[36] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Evaluation of design options for the trace cache fetch mechanism," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 435–446, February 1999.

[37] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Critical issues regarding the trace cache fetch mechanism," Technical Report CSE-TR-335-97, University of Michigan Technical Report, May 1997.

[38] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark, "One billion transistors, one uniprocessor, one chip," *IEEE Computer*, vol. 30, pp. 51 – 57, September 1997.

[39] Y. Patt, W. Hwu, and M. Shebanow, "HPS, a new microarchitecture: Rationale and introduction," in *Proceedings of the 18th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 103–107, 1985.

[40] Y. N. Patt, S. W. Melvin, W. Hwu, and M. C. Shebanow, "Critical issues regarding HPS, a high performance microarchitecture," in *Proceedings of the 18th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 109–116, 1985.

[41] A. Peleg and U. Weiser. *Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independant of Virtual Address Line.* U.S. Patent Number 5,381,533, 1994.

[42] K. Pettis and R. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 16–27, 1990.

[43] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.

[44] E. Rotenberg, Q. Jacobsen, Y. Sazeides, and J. E. Smith, "Trace processors," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[45] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block ahead branch predictors," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[46] R. L. Sites, *Alpha Architecture Reference Manual*, Digital Press, Burlington, MA, 1992.

[47] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 135–148, 1981.

[48] M. Smotherman and M. Franklin, "Improving CISC instruction decoding performance using a fill unit," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 219–229, 1995.

[49] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, "The agree predictor: A mechanism for reducing negative branch history interference," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[50] E. Sprangle and Y. Patt, "Facilitating superscalar processing via a combined static/dynamic register renaming scheme," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 143–147, 1994.

[51] *Welcome to SPEC*, The Standard Performance Evaluation Corporation. http://www.specbench.org/.

[52] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[53] S. Vajapeyam and T. Mitra, "Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 1–12, 1997.

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

[55] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive branch prediction," in *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 51–61, 1991.

[56] C. Young, N. Gloy, and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 276–286, 1995.