# Characterizing and Removing Branch Mispredictions

Kevin Skadron

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

June 1999

# Abstract

Control-flow mispredictions are a profound impediment to processor performance, because each misprediction introduces a pipeline bubble of many cycles' duration. For example, the minimum bubble in the recently released Alpha 21264 is at least seven cycles, and often as much as twenty cycles. With such long penalties, even small misprediction rates harm performance substantially.

Although a huge number of techniques have been proposed to combat this problem, most have focused on only one type of misprediction: conflicts in the predictor's state tables.

This thesis describes a taxonomy of misprediction types and presents data showing that several other types of mispredictions are just as important as conflicts. Techniques to attack three of these misprediction types are then described. *Alloying* makes both local and global history available in a single branch predictor structure, providing robust performance compared against both conventional two-level predictors and against hybrid predictors. *Speculative update with fixup* ensures that the predictor's state remains up-to-date, while protecting the state against corruption from mispredicted branches. Finally, *multipath execution* simultaneously executes both sides of a branch, eliminating mispredictions for those branches that are otherwise difficult to predict.

This work also explores tradeoffs among branch prediction, instruction window size, data cache size, and instruction cache size, and shows that branch prediction is the most powerful lever on processor performance.

This thesis makes one further contribution, demonstrating that long benchmarks can be simulated efficiently by simulating only a small but carefully chosen 50 million instruction segment of the program's overall execution. This technique avoids un-

representative initial phases of a program and sets the simulation window to capture behavior that is representative in terms of the program's branch prediction accuracy, cache behavior, and overall execution speed.

# Acknowledgments

I am fortunate to have had not one, but two excellent advisors, Doug Clark in the Department of Computer Science, and Margaret Martonosi in the Department of Electrical Engineering. I have learned a great deal from both of them, and they have staunchly backed me through thick and thin. Gary Tyson, in the University of Michigan's Department of Electrical Engineering and Computer Science, and Kai Li, in Princeton's Department of Computer Science, have also been immensely supportive and have provided valuable advice. J.P. Singh and Ed Felten also provided frequent advice and were kind enough to serve on my dissertation committee.

I am also fortunate to have had, while an undergraduate, the advice and support of Willy Zwaenepoel, a professor in Rice's Department of Computer Science. He and my former roommate at Rice, Steve Rodrigues, were instrumental in launching my computer science career. Without their mentorship I would most likely not have attended graduate school!

I am indebted to many additional people for helping me finish this dissertation. The SHRIMP group provided me with numerous computers for my hungry simulations, and Angelos Bilas, Rudro Samanta, and Yuanyuan Zhou in particular bent over backwards to accommodate my needs. The "CS Staff" also deserve a special thanks for putting up with my heavy use of the department's compute servers, and with my incessant questions. This research would not have been possible without access to all these computers.

Pritpal Ahuja helped me with numerous technical and simulation questions, and was a regular sounding board for me. Others to whom I owe technical debts include Todd Austin (University of Michigan), John Bennett (Rice University), Adam Finkelstein (Princeton), Konrad Lai (Intel MRL), Andrea LaPaugh (Princeton), Sally

McKee (University of Utah), Anne Rogers (AT&T Research), Wen-Hann Wang (Intel MRL), and Chris Wilkerson (Intel MRL). Rich Alpert (NEC) helped me preserve my sanity during the general exam process, and spent many hours discussing thesis and career issues with me. George Busenberg, another former roommate at Rice, has been and continues to be a reliable source of advice and wisdom. And finally, Ginny Hogan, Melissa Lawson and Trish Zabielski helped me survive all manner of administrivia and provided much advice on how to make the most of my life in Princeton.

To my office mates, both past and present, I offer thanks for tolerating me for five years: George Chesakov, Stef Damianakis, Haim Kaplan, Sanjeev Kumar, Lena Petrovic, George Tzanetakis, Limin Wang, and Jiannen Zheng. I have made many good friends at Princeton, too many to name here, but in particular I want to mention Pritpal Ahuja, Jessica Baraka, Dan Boneh, Catherine Brown, Pei Cao, Yuqun Chen, Stef Damianakis, Wenjia Fang, Jason Faris, Britta Gilmore, Helen Hu, Scott Karlin, Jeff Korn, Paul Shocklee, Elizabeth Stuart, Bob Thomas, Dan Wallach, and Diane Whitmore, all of whom provided valuable friendship and support.

Most importantly, I thank my family for their support and for making this possible: my parents, my grandparents, my Aunt Eileen, and my fiancée, Rachel Berez.

Funding support for this work was provided by a Department of Defense NDSEG fellowship and by NSF grants CCR-94-23123 and EIA MIP94-20653.

To my family, who made this possible

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

This thesis shows that *branch prediction* is a powerful lever over processor performance, and then describes a taxonomy of the possible sources of branch mispredictions. Using this taxonomy, the thesis shows that several little-studied sources of mispredictions are in fact important contributors to the overall misprediction rate. Techniques for reducing three of these misprediction types are then described.

Branch prediction accuracy is important because it underpins microprocessor performance. That, in turn, underpins the performance of a spectrum of computer systems, from simple handheld devices to large multicomputers. Branches arise from `if` statements, `for` loops, and so forth, and are the instructions that allow a program to control the sequence of computations performed by choosing between two possible actions. This presents a problem because branches take time to execute and resolve: at least seven clock cycles in the recently-released Compaq Alpha 21264 [32, 49], and at least eleven clock cycles in the Intel Pentium Pro and its successors [31]. The identity of the next instruction to be executed can only be known after this resolution process has completed.

The time lost to branch resolution delays can be attacked on several fronts. Processor designers can attempt to reduce the branch resolution time, but this has proven difficult, and necessitates special-case hardware that complicates the overall design. Indeed, branch-resolution times have been increasing. Furthermore, even if this time is reduced, every branch still incurs the stall time which cannot be eliminated. Instead, the compiler can reduce the number of branches executed using techniques like predication [2, 75] or trace scheduling [23]. This requires sophisticated compiler analysis, and usually treats each branch instruction the same way every time that instruction is encountered. Yet branches may behave differently over the course of the program as program conditions or input data change. Yet another alternative is for the programmer to eliminate branches by changing the program's algorithm. This process, however, cannot generally be automated, and requires painstaking analysis for each program.

An especially attractive approach to reducing the time spent on branches is to predict branch outcomes. Then, while the branch resolves, instructions from the predicted path can be executed speculatively. If the prediction succeeds, the branch's resolution time has been successfully hidden with useful computation. If the prediction fails, the speculative instructions must be squashed, but otherwise no harm has been done. Prediction does not preclude any of the techniques mentioned above, and in fact, the different approaches are complementary. Prediction has the distinct advantage that it does not change the program and risk slowing down its execution; prediction is entirely transparent to the programmer.

Without prediction, each branch that has not been removed introduces some stall time into the program's execution. With prediction, branches only introduce stall time on a misprediction. Prediction can be done statically using a combination of compiler analysis and profile information gathered from previous executions, or it can be done dynamically using dedicated prediction hardware in the processor. Dynamic prediction requires some allocation of hardware resources on the processor for the predictor structures that track branch behavior. But dynamic techniques can exploit information that is only available at runtime, and detect branch behavior that varies over the course of a program's execution. This thesis focuses on dynamic branch prediction techniques.

Branch prediction has been a vibrant research topic for more than ten years, because the stalls that result from remaining mispredictions constitute one of the key stumbling blocks to continued improvements in processor performance. These misprediction latencies are growing longer. Whereas RISC processors from the late 1980s and early 1990s had latencies of only one or two cycles, modern processors like the 21264 and Pentium Pro have *minimum* latencies of seven to eleven cycles. The *average* latency is often longer yet in modern processors like these, because instructions can experience arbitrary delays during execution, making the average latency 10–20 cycles. With such long delays, it takes only a tiny misprediction rate to add a substantial amount of stall time. Consider, for example, an average branch frequency of one out of every six instructions, fairly typical of non-scientific, integer-based programs like compilers or simulators, and a mere 5% misprediction rate. A simple, single-issue processor with a ten-cycle misprediction latency and no other sources of delay experiences an 8% slowdown. Yet many programs have much worse prediction accuracies, with misprediction rates as high as 15–20%.

Compounding the continuing problem of misprediction latencies is the fact that processors are moving to wider issue widths. Processors in the early 90s generally issued one instruction per cycle. Today's processors issue up to six instructions per cycle (*e.g.*, the Alpha 21264), and future processors may use pursue even wider issue widths. This is especially likely in VLIW processors, in which there is growing interest. As processors become wider, each cycle of misprediction latency wastes more issue opportunities, increasing the degree to which each misprediction penalizes performance. In the same circumstances for which a single-issue processor experiences an 8% performance penalty, a processor with four-way issue may experience as much as a 32% penalty.

In addition to the direct problem of misprediction penalties, branch mispredictions also impede performance indirectly, interfering with the operation of other processor structures, like the data-value predictor or the instruction window.

The next chapter describes my simulation methodology, and then Chapter 3 examines in further detail how branch mispredictions impede performance, by exploring how branch prediction improvements trade off against improvements in instruction window size and cache size. The results show that among these design parameters, branch prediction accuracy is the most powerful lever over the processor's performance. A similar study by Ranganathan and Jouppi [42] explores trends in branch prediction and main-memory design, and finds that aggressive but feasible techniques for reducing memory latency and bandwidth can prevent these factors from becoming substantial bottlenecks. In other words, they argue that no conceptual breakthroughs are needed to alleviate these problems. In contrast, they find that branch mispredictions are already a severe problem, and given the trends toward increasing misprediction latencies and wider issue widths, this problem will only grow worse. They conclude that without breakthroughs in branch prediction techniques, mispredictions are likely to become the dominant performance bottleneck by the year 2010.

Once the importance of continuing improvements in branch prediction has been demonstrated, this thesis explores a new approach to the problem of designing more accurate dynamic predictors. Most past work has proposed one general predictor mechanism designed to improve upon and replace prior designs. Yet, as I will show, branch mispredictions happen for a number of distinct causes. In Chapter 4, I describe a taxonomy of misprediction causes and measure the contribution of each to the overall misprediction rate. This in turn permits the design of solutions tailored to reduce each important category of mispredictions. These solutions can then operate in parallel, or better yet, can be composed into a new predictor. The latter is true for the solutions proposed here. Chapter 4 also describes and evaluates *alloying*, which reduces the likelihood that the branch predictor is tracking the wrong type of branch history. Chapter 5 explores the update timing for the branch predictor, shows how stale history can be eliminated by updating the predictor speculatively, and describes fixup mechanisms for eliminating any corruption that the speculative update might introduce. Chapter 6 explores similar issues for the return-address stack. Chapter 7 describes a technique for eliminating mispredictions associated with intrinsically hard-to-predict branches, *multipath execution*. This technique speculates on both possible outcomes of a branch. This eliminates mispredictions without the need to use a predictor on these difficult branches. Finally, Chapter 8 summarizes the thesis and outlines several avenues for future work.

The remainder of this chapter presents the background necessary to prepare the reader for the remaining chapters, and can safely be skipped by those generally familiar with branch prediction. Section 1.1 explains how branches are executed in a typical processor and elaborates on the reasons for branch resolution delays, and Section 1.2 describes the types of branch predictors used in current processors. Finally, Section 1.3 surveys some branch prediction research of general interest; work related specifically to the topics explored in later chapters will be discussed within those chapters.

## 1.1 Anatomy of a Branch

Branch instructions permit a program to control what instructions are executed. Examples of program-level constructs that use branches are `if-then` statements and `for` loops. They test some condition, and depending on the outcome, execution proceeds down one of two possible paths. In almost all instruction sets, branch instructions have exactly two possible outcomes: *not-taken*, the sequential case in which the program continues executing the instructions that immediately follow the branch, and *taken*, the non-sequential case in which execution jumps to a target specified in the branch instruction. This target can lie anywhere within the program. Multiway conditional statements, like the `switch` statement in C, are composed of a sequence of branches.

In the case of an `if` statement, these two branch outcomes correspond to the `then` clause and to the fall-through case (which often corresponds to an `else` clause). In the case of a `for` loop, the two possibilities are an iteration of the loop body or the fall-through case which terminates the loop.

As mentioned earlier, branches create a problem because the identity of the proper path can only be known after the branch has finished testing its condition, a process that takes time. Due to the pipelined nature of almost all modern processors, this resolution latency necessitates branch prediction. Figure 1.1 shows the flow of a branch through a generic pipeline. Resolving the branch requires waiting until it proceeds through several stages. If the fetching of subsequent instructions must wait until the proper path is known with confidence, stall time or a "bubble" results.

If branch outcomes are instead predicted and subsequent instructions are speculatively fetched and executed, this bubble is eliminated whenever the prediction is correct. This is shown in Figure 1.2. If the prediction is incorrect, these speculative

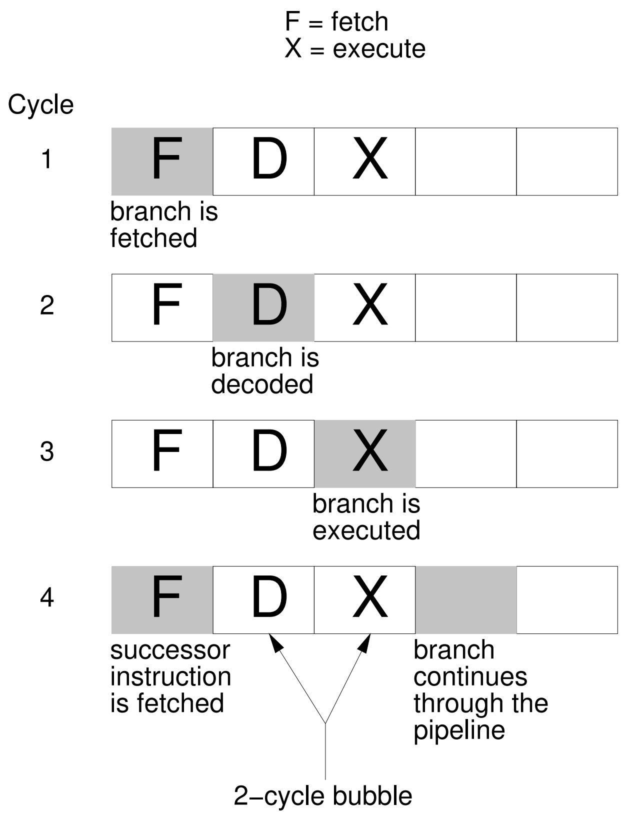


Figure 1.1: A branch flowing through a generic pipeline with no prediction. The branch, the first gray box, flows from left to right. After being fetched, one or more cycles elapse (one cycle in this diagram) while the instruction is decoded and perhaps other manipulation takes place. Once the branch finally completes executing (*i.e.*, testing its condition), the next instruction (the next gray box) can be identified and fetched. This figure shows that the resolution time introduces a delay during which the pipeline stalls. The corresponding “bubble” here is 2 cycles long.

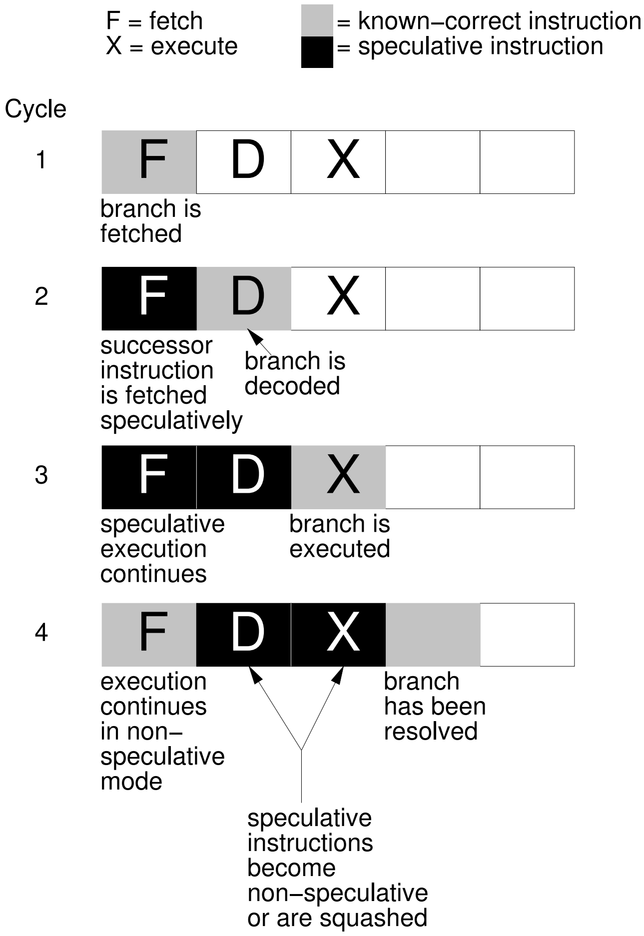


Figure 1.2: Pipeline behavior with branch prediction.
In this diagram, the branch's outcome is predicted. Immediately in the next cycle, subsequent instructions are fetched and executed speculatively (black boxes). If the prediction is correct, the speculative instructions do useful work and the bubble has been eliminated. If the prediction is incorrect, the speculative instructions are squashed.

instructions must be squashed—removed from the pipeline—and no time has been wasted compared to the alternative of no prediction. Squashing can be accomplished simply by preventing the mis-speculated instructions from modifying any processor state.

Other control-flow instructions exist that transfer execution to some other program location but are not conditional and do not branch. These *jump* instructions either jump to the target specified in the instruction (direct jumps), or jump to a target that has been computed and whose address is found in a register (indirect jumps). A procedure call is an example of the former, and a procedure return is an example of the latter. As with branches, some time is required to determine jump targets. Direct jumps can be resolved early with proper hardware in the fetch stage to extract the jump target from the instruction, or the targets can be predicted. Indirect jumps generally cannot be resolved early, and instead must proceed through the pipeline in order to read their target from the register file, just like any other instruction. Fortunately, their targets can also be predicted. Prediction of indirect branches is an active research topic [9, 12, 15, 47], but is beyond the scope of this thesis. Return instructions are a special case of indirect jumps, and are easily predicted using a simple structure known as a *return-address stack* [46, 108]. Chapter 6 extends Chapter 5's exploration of update timing for conditional-branch prediction to the return-address stack, but otherwise this thesis focuses on prediction of conditional branches.

## 1.2 Basic Branch Prediction Techniques

A number of modern processors include dynamic branch predictors, and this section describes the basic types of predictors that have appeared: bimodal, two-level, and hybrid.

### 1.2.1 Bimodal Prediction

The most basic mechanism is *bimodal prediction*, a simple table of binary values, one per branch. This value is updated to always provide the last outcome witnessed for each branch, and so each time a branch changes direction from taken to not-taken or vice-versa, a misprediction results. Some of these mispredictions can be eliminated by making the table entries two-bit saturating counters. Each not-taken outcome decrements the counter (until it hits 00), and each taken outcome increments it (until it hits 11). Values of 00 and 01 produce a not-taken prediction, and values of 10 and 11 produce a taken prediction. The two-bit counters give the table entries some degree of hysteresis, reducing the number of mispredictions when branches change direction for only a single occurrence. For example, a loop branch will be mispredicted twice with a one-bit counter: once when the loop exits, and once more when the program re-enters the loop. A two-bit counter recognizes that the common case is for the loop to iterate (a taken outcome), and in the steady state the counter contains a value of 11. When the loop exits, the branch will be mispredicted and the counter decremented to 10. When the loop is encountered again and resumes iterating, the first iteration sees the value 10 and predicts taken, even though the most recent outcome was a not-taken outcome. Both bimodal schemes were proposed by J. Smith [90]. A number of processors have included two-bit bimodal predictors, including the Alpha 21064 [13] and 21164 [14], the MIPS R10000 [113], and the Sun UltraSPARC I [28].

### 1.2.2 Two-Level Prediction

More bits can be added to the two-bit counters in the bimodal scheme, but too many bits make it difficult for the predictor to learn legitimate changes in direction. A better refinement is to explicitly track branch history patterns, and for each branch, make different predictions depending on the recent history. These predictors are called *two-level predictors*. A *local history* predictor keeps a table of shift registers, one entry per branch. A typical history width is ten bits. To make a prediction, the predictor looks up the branch's history, and then uses that history to index the now-familiar table of two-bit counters. These counters now track the taken/not-taken behavior of branch history patterns, and not the overall behavior of individual branches. This permits common patterns, like alternating branches (TNTN...), to be correctly predicted, and can also learn irregular patterns that correspond to some program or input data behavior (*e.g.*, TTTNTTTN...). An example of a local history predictor appears in Figure 1.3.

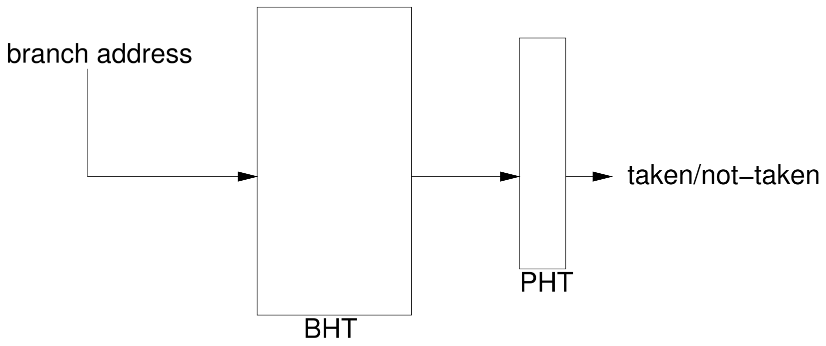


Figure 1.3: A local-history-based two-level predictor.

Alternatively, the table of per-branch histories (the *branch history table* or BHT) can be replaced with a single, global shift register, the *global branch history register* or GBHR. All branches shift their outcomes into this register, which typically is 10–15 bits wide. This may seem a strange thing to do, but *global history* prediction allows branches to easily see the behavior of other recent branches. This is useful for sequences of correlated branches, like

```
B1:        if (A)
              ...
B2:        if (B)
              ...
B3:        if (A and B)
              ...
```

Even if branches B1 and B2 are unpredictable, branch B3 is 100% predictable if the outcomes of B1 and B2 are available, something that global history prediction accomplishes. Global history is also effective when the GBHR's string of global-history bits uniquely identifies a branch, regardless of whether it exposes correlation.

A typical global history predictor appears in Figure 1.4. Both types of two-level prediction were proposed concurrently by Yeh and Patt [114] and Pan, So, and Rahmeh [70].

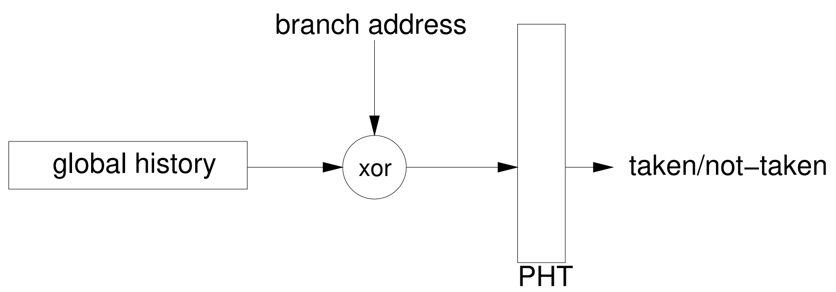


Figure 1.4: A global-history-based two-level predictor.

Figure 1.4 also shows the inclusion of some address bits in the index for the table of two-bit counters (the *pattern history table* or PHT). The preceding discussion has assumed each branch has a unique entry in these tables, but these are hardware structures and necessarily of finite size. Two branches may therefore share the same entry, either because the table is not sufficiently large or because the two branches share the same prior history. This may be harmless, and sometimes even helps when it accidently permits related branches to communicate additional information among each other. But if often results in destructive interference as the branches overwrite each others' state. This aliasing can be alleviated in both global and local history predictors by combining the history bits with some bits from the branch's address. One simple way to do this, proposed by McFarling [64] is to XOR the two patterns together, creating a *gshare* predictor. Now branches that share the same history are usually distinguished by their different addresses.

### 1.2.3 Hybrid Prediction

Within most programs, some branches are best predicted using global history, while others are best predicted using local history. A processor that only implements one or the other type of predictor therefore penalizes some branches.

A *hybrid predictor* includes multiple predictors [20, 64], with some way to choose which predictor to use at any given time. A simple and effective organization is a two-component hybrid predictor that combines a global history and a local history predictor [11]. Figure 1.5 shows such a predictor.

The selection mechanism can be static, with branches assigned to a particular component at compile time [30], or dynamic, with a hardware structure that learns when to use the different components. The dynamic predictor looks like another branch predictor, but instead of tracking branch outcomes, tracks predictor success. With two predictor components, the dynamic selector looks like either a bimodal predictor [64] or a two-level predictor [11].

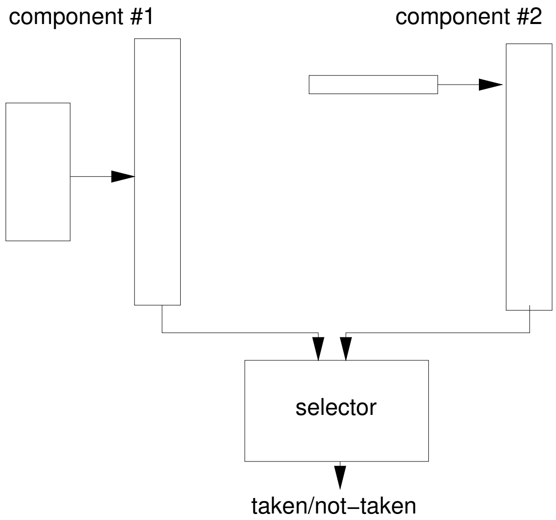


Figure 1.5: The organization of a hybrid predictor with two components. If the hybrid predictor statically assigns branches to one or the other component, the selector simply chooses the appropriate component depending on the branch's PC.

## 1.3 Related Work

As mentioned at the beginning of this chapter, the compiler can reduce the impact of conditional branches by entirely eliminating some of them. Unfortunately, control dependencies are often hard to remove outright, so predication [2, 75] is an alternative compiler optimization that removes conditional branches by converting them into conditional moves. This process is also sometimes referred to as *if-conversion*. The dependence on the condition remains, but it is now a data and not a control dependence. This moves the problem to a later point in the pipeline—which is particularly effective with out-of-order execution—and allows the fetch engine to fetch straight-line code. Recent work has characterized how extensively predication should be used [60], explored techniques for better analysis of predication opportunities [40], and explored the relationship between predication and branch prediction efficacy [59, 72, 101].

Compiler analysis also permits static branch prediction. When combined with profiling and/or branch-history information like the history that dynamic two-level predictors use, the compiler can often do an effective job of statically predicting branch outcomes, as shown by Young, Gloy, and Smith [116, 117] and Fisher and Freudenberger [24]. Fisher and Freudenberger also pointed out that the number of instructions between mispredictions is a better indicator of a branch predictor's performance than the more common metric of overall misprediction rate, which only measures the fraction of dynamic branches that mispredict. The former provides information about the average distance between mispredictions and therefore their potential impact on performance, while the latter does not.

Profiling also permits trace scheduling [23], which assumes the branch will behave according to the static prediction and simply elides it, creating straight-line code. Trace scheduling is closely related to procedure inlining [36], which replaces procedure calls with copies of the procedure bodies, again creating straight-line code. The difference is that inlining is a safe operation, and the only question is whether the consequent code expansion is worthwhile. Trace scheduling, however, is speculative, and still requires instructions to test the condition and make the appropriate corrections if the assumed outcome proves false.

This thesis focuses on dynamic branch prediction mechanisms, already a subject of numerous papers. Only some broad-based papers are mentioned here—those that explore the circumstances under which various mechanisms work well—while related work specific to the chapters in this dissertation will be discussed within those chapters. Almost all mechanisms include a table of two-bit saturating counters—the PHT—and most work has focused on improving the behavior of this table by considering whether the PHT should in fact use dynamic counters [83], and by showing the importance of aliasing in the PHT as a contributor to the misprediction rate [57, 66, 82, 94]. Kim and Tyson [50] examine the impact of aliasing in the table of per-branch histories—the BHT—and find it is a further important contributor of mispredictions in local history predictors. Yeh and Patt [115] explore the design space for both global history and local history predictors of different sizes, and find that the most effective overall configuration and even the most effective history type vary widely according to the hardware budget available. Evers, Patel, Chappell, and Patt [21] study the operation of branch histories to better understand when they successfully identify inter-branch correlation and repeating patterns, and when they fail to recognize branches that are nevertheless predictable. They find that idealized forms of correlation substantially outperform existing mechanisms—current history schemes are not optimal—and then show that in a global history predictor, this happens because the history bits contained in the GBHR at any particular time are often not the most useful bits of information. Finally, a novel paper by Emer and Gloy [19] should be mentioned. Their work "breeds" new types of branch predictors using genetic mutation algorithms. While the resulting mechanisms are usually complex and not realistic to implement, the individual components tend to resemble many aspects of existing dynamic branch predictors, adding further evidence that the mechanisms described in the preceding section, like two-level predictors, are indeed suitable solutions.

A general survey of branch prediction techniques, including examples from commercial processors, can be found in [27], and a more recent survey can be found in [105]. In particular, this second article argues for new, unconventional techniques for eliminating branch mispredictions, and focuses on multipath execution, a different form of which this thesis explores in Chapter 7.

# Chapter 2

# Experimental Methodology

This section describes the simulation infrastructure used to conduct the studies in this dissertation, and the SPEC95 benchmark programs [98] used to obtain measurements.

The SPEC programs run for billions or even hundreds of billions of cycles. Because cycle-accurate simulators take a long time to simulate such programs to completion—days or weeks for some of the benchmarks—I have developed a way to select a single, 50 million (M) instruction segment of each program's execution. Careful selection of this window can assure that the captured behavior is representative in terms of branch misprediction rate, cache miss rates, and overall performance (in terms of IPC or *instructions per cycle*). In particular, this technique avoids unrepresentative startup behavior from early phases of a program's execution. The last section in this chapter describes how to choose a simulation window, and presents some data strongly suggesting that these small simulation windows indeed capture representative behavior.

## 2.1 Simulator

Most of the data presented in this thesis have been obtained using *HydraScalar*, my heavily modified version of SimpleScalar's [5] *sim-outorder* simulator. SimpleScalar provides a toolbox of simulation components—like a branch predictor module, a cache module, and a statistics-gathering module—as well as several simulators built from these components. Each simulator interprets executables compiled by *gcc* version 2.6.3 for a virtual instruction set called the "PISA" that most closely resembles MIPS IV [74]. The simulators emulate the executable's execution in varying levels of detail.

HydraScalar is a cycle-accurate simulator. Chapter 4 uses a simpler, instruction-level simulator to reduce simulation costs, because the work there is less sensitive to cycle-by-cycle effects. Cycle-by-cycle simulators like HydraScalar that do their own instruction fetching and functional simulation (as opposed to relying on direct execution to provide instructions for simulation) can accurately simulate execution down mis-speculated paths. Like a real processor, HydraScalar checkpoints appropriate state as it encounters branches, and then upon detecting a mispredicted branch, wrong-path instructions are squashed, and the correct state is recovered using the checkpointed state. Modeling the actual instruction flow on mis-speculated paths captures consequences like prefetching and cache pollution.

HydraScalar models in detail an out-of-order execution (OOE), five-stage pipeline: fetch (including branch prediction), decode (including register renaming), issue, write-back, and commit. An arbitrary number of further stages can be added between decode and issue to simulate time spent renaming and enqueuing instructions (the experiments in this thesis add three extra cycles). Issue selects the oldest ready instructions.

Table 2.1 summarizes the baseline model used for most of this work, which loosely resembles the configuration of an Alpha 21264 [49]. The simulations use a two-level, non-blocking cache hierarchy with miss-status holding registers (MSHRs) in both the first-level (L1) and second-level (L2) caches [53]. The cache module simulates a simple pipelined bus with fixed fetch spacing—the bus can accept a new transaction every $n$ cycles, and once started the transaction completes without contention—and fixed memory latency (no interleaving, reordering, or fast-page-mode accesses). The model also assumes perfect write buffering (stores consume bandwidth but never cause stalls), which should have minimal impact on these results, especially for non-numeric codes [86]. HydraScalar simulates a unified active list, issue queue, and rename register file. This type of instruction window is called a *register update unit* or RUU [93]. The architectural register files (32 registers each for integer and floating-point) are separate and updated on commit. Using an RUU eliminates artifacts arising from interactions between active-list size and issue-queue size. A load-store queue (LSQ) disambiguates memory references: stores may only pass preceding memory references whose addresses are known not to conflict. This work does not explore LSQ size, so for simplicity the LSQ is fixed at one-half the RUU size to eliminate artifacts from stalls should the LSQ fill up.

Many entries in the PHT (the pattern history table consisting of the saturating, two-bit counters) correspond to not-taken branches (or are simply idle). For this reason, an entry is only allocated in the *branch target buffer* (BTB) for taken branches. This permits the BTB to have fewer entries than the PHT [7]. The fetch stage modeled by HydraScalar makes a prediction for each branch fetched, but within a group of fetched instructions, those following the first predicted-taken branch are discarded because control must now jump to a new location. In practice, therefore, the fetch engine fetches through not-taken branches and stops at taken branches.

Different branch predictors are simulated in this dissertation, and the configu-

| Parameter | Value | Comments |
|---|---|---|
| Processor core | | |
| RUU size | 64 | Instruction window |
| LSQ size | 32 | Enforces load-store ordering |
| Decode latency | 4 cycles | Min time b/t fetch and issue |
| Fetch width | up to 4 instructions per cycle | Must be in same cache block |
| Decode width | up to 4 instructions per cycle | In-order |
| Issue width | up to 4 integer ops per cycle plus 2 FP ops per cycle | Out-of-order |
| Commit width | up to 4 instructions per cycle | In-order |
| Functional units | 4 ALU/logical (1), 2 branch/shift(1), 1 integer multiply/divide (12/20), 1 FP add (4), 1 FP multiply (4), 1 FP divide/sqrt (16/33) | Latency appears in parentheses, and does not include the 1 cycle for reading registers |
| Memory ports | 2 | any combination of loads or stores |
| Branch prediction | | |
| BTB | 2048-entry, 2-way | updated only if taken |
| Return-addr. stack | 32-entry | repaired after mis-speculation |
| Mispredict penalty | 7 cycles | 2 cycles for BTB-miss only |
| Memory hierarchy | | |
| L1 D-cache | 64 K, 2-way (LRU), 32 B blocks | 8 MSHRs, 1-cycle latency |
| L1 I-cache | 64 K, 2-way (LRU), 32 B blocks | 2 MSHRs, 1 cycle latency |
| L2 | unified, 8 M, 4-way (LRU), 32 B blocks | 4 MSHRs, 12-cycle latency |
| Memory | 100 cycle latency; 8 B/cycle | |
| L1→L2 bus | 1 transaction every 2 cycles | |
| L2→mem bus | 1 transaction every 8 cycles | |
| TLBs | 128 entry, fully assoc., 30-cycle miss latency | both I-TLB and D-TLB |

Table 2.1: Baseline configuration simulated by HydraScalar.

rations used in each chapter are described there. Regardless of the configuration, HydraScalar updates the predictor state during the instruction-commit stage, except where noted. This means there is a window of time, while a branch traverses the pipeline, during which its outcome is not available and the branch predictor uses slightly "stale" state. The resulting prediction accuracies are thus not as high as those reported in trace-driven prediction studies that do not model cycle-by-cycle timing effects. Although timing and implementation details of commercial microprocessors are difficult to come by, it seems that early—*i.e.*, speculative—history update is only a recent innovation because it requires fixup mechanisms to undo corruption from incorrect updates. The importance of speculative update is discussed in [33], and mechanisms for implementing fixup appear in [44] and Chapter 5. The only known example of speculative update appears in the Alpha 21264, which implements speculative update with fixup for just the global-history portion of its hybrid predictor [49].

Many mispredicted indirect jumps are function returns. Since a function might be called from many different locations, a BTB often provides the wrong target for these jumps (it only remembers the last target). A return-address stack is a natural solution. It is best pushed and popped speculatively, in the fetch stage, and thus requires a fixup mechanism to prevent corruption. HydraScalar models a simple mechanism, described later in Chapter 6, that does not guarantee restoration of the correct state, but in practice virtually eliminates corruption. All the simulations reported in this thesis use this mechanism and, like the Alpha 21264, use a 32-entry stack [49].

Branch direction mispredictions suffer at least a seven-cycle latency, because the branch condition is not resolved until the writeback stage. The latency may be longer if the branch spends time waiting in the RUU. Conditional jumps for which the predicted direction is correct—and direct jumps—can still miss in the BTB (a *misfetch*), but a dedicated adder in the decode stage computes branch targets so that BTB misses can be detected early. A BTB miss therefore only experiences a two-cycle penalty. Indirect jumps need to read the register file, which is at best expensive to do from decode and is therefore not supported in these models. When the BTB mispredicts these targets, the error is only detected in the writeback stage.

## 2.2 Benchmarks

This research focuses on the SPEC95 integer benchmarks [98], summarized in Tables 2.3 and 2.4, and uses the provided SPEC95 "reference" inputs. Like any other suite, SPEC has its shortcomings: for example, many of the programs fit in small instruction caches. Because commercial workloads differ in some ways from the SPECint programs [63], users should exercise care in translating this thesis's results to such programs: the trends and pitfalls identified here should hold true for many programs, but clearly not for all. Another possible pitfall is that this work excludes operating-system calls, because SimpleScalar does not not support that and because it adds substantially to simulation times. Some program behavior—including branch behavior—does vary when operating system calls are included [26]. Fortunately, this is not a problem for most SPEC benchmarks, as they call the operating system only rarely [26]. Furthermore, the simulation lengths used here are short and on the order of context-switch intervals. The next section describes how these short simulation windows are selected.

The integer benchmarks are the primary focus of this work, because the floating-point programs have near-perfect branch prediction accuracies, lower dynamic branch frequencies, and small text sizes. Some measurements do include *tomcatv* as a reference for how floating-point programs might behave. Chapter 4 adds two other SPECfp and four non-SPEC integer benchmarks; these additional benchmarks are described in that chapter. All benchmarks are compiled using `gcc -O3 -funroll-loops` (-O3 includes inlining). Tomcatv is first translated to C using AT&T Bell Labs' *f2c* (1994 version). Simulations include all non-kernel behavior, like library code.

Some benchmarks come with multiple reference inputs, in which case only one has been chosen. *Go* is run with a playing level of 50 and a 21x21 board with the `9stone21` input. For *m88ksim*, the `dhrystone` input is used; for *gcc*, `cccp.i`; for *ijpeg*, `vigo.ppm`; and for *perl*, the scrabble game is used. Unlike these benchmarks, *xlisp* is run with all the supplied LISP files as arguments, because each individual file is small.

As mentioned, full-detail simulation is only performed for a representative 50 million instruction segment of the program. This window is carefully selected to capture behavior that is representative in terms of branch misprediction rate, cache miss rates, and overall IPC, and in particular, to avoid unrepresentative startup behavior. The next section presents a detailed discussion of how these windows are chosen. Cycle-level simulations are run in a fast mode to reach the chosen simulation window. In this "warmup" phase, no pipeline simulation takes place; only caches, branch predictor, and architectural state are updated with each instruction. Then one million instructions are simulated in full detail to prime other structures. Table 2.2 presents the total length of the warmup phase. Table 2.3 summarizes branch behavior during the simulation window for each benchmark, and Table 2.4 summarizes the memory reference behavior.

Even though a branch can move from fetch to writeback in eight cycles in our model, branches must wait for operands and then arbitrate for issue. In fact, branches typically take ten or more cycles to resolve. For reference purposes, Table 2.5 presents average branch-resolution times and their standard deviations. Note that the delay

| | Warmup Insts | | Warmup Insts |
|---|---:|---|---:|
| go | 926 M | ijpeg | 824 M |
| m88ksim | 26 M | perl | 601 M |
| gcc (cc1) | 221 M | vortex | 2451 M |
| compress | 2576 M | tomcatv | 2276 M |
| li (xlisp) | 271 M | | |

Table 2.2: Benchmark summary—simulation window size (millions of instructions)

| | Warmup Insts | Branches per Instruction | | | | Branch Accuracies | | | |
|---|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| | | All | Return | Indir | Cond | All | Return | Indir | Cond |
| go | 926 M | 0.144 | 0.011 | 0.002 | 0.111 | 0.758 | 0.432 | 0.629 | 0.754 |
| m88ksim | 26 M | 0.212 | 0.018 | 0.003 | 0.162 | 0.931 | 0.704 | 0.251 | 0.954 |
| gcc | 221 M | 0.194 | 0.015 | 0.030 | 0.144 | 0.827 | 0.546 | 0.350 | 0.861 |
| compress | 2576 M | 0.202 | 0.028 | 0.000 | 0.133 | 0.925 | 0.993 | 0.063 | 0.888 |
| li (xlisp) | 271 M | 0.236 | 0.027 | 0.082 | 0.137 | 0.906 | 0.721 | 0.814 | 0.918 |
| ijpeg | 824 M | 0.059 | 0.001 | 0.003 | 0.051 | 0.894 | 0.728 | 0.984 | 0.879 |
| perl | 601 M | 0.193 | 0.019 | 0.077 | 0.129 | 0.893 | 0.664 | 0.332 | 0.937 |
| vortex | 2451 M | 0.166 | 0.021 | 0.021 | 0.121 | 0.968 | 0.901 | 0.768 | 0.980 |
| tomcatv | 2276 M | 0.192 | 0.000 | na | 0.130 | 0.999 | na | na | 0.999 |

Table 2.3: Benchmark summary—branch statistics.

Statistics are taken only from the post-warmup, 50 M-committed-instruction simulation window, and use the baseline configuration in Table 2.1. "All" refers to all branches, whether conditional, direct-jump, indirect-jump, or return. "Indirect branches" here does not include returns. "Branch accuracy" refers to target-address prediction, except for the conditional-branch column, which presents direction prediction accuracies.

| Benchmark | Mem Refs per Inst | Miss Ratios I$ | Miss Ratios D$ | Miss Ratios L2 |
|---|---|---|---|---|
| go | 0.299 | 0.004 | 0.005 | 0.000 |
| m88ksim | 0.301 | 0.000 | 0.000 | 0.000 |
| gcc | 0.453 | 0.008 | 0.010 | 0.000 |
| compress | 0.341 | 0.000 | 0.084 | 0.003 |
| xlisp | 0.405 | 0.000 | 0.016 | 0.000 |
| ijpeg | 0.271 | 0.000 | 0.008 | 0.000 |
| perl | 0.465 | 0.001 | 0.003 | 0.000 |
| vortex | 0.526 | 0.009 | 0.004 | 0.000 |
| tomcatv | 0.211 | 0.000 | 0.035 | 0.020 |

Table 2.4: Benchmark summary—cache statistics.
L2 cache miss rates are global, that is, over all references made by the program.

| | average | std-dev (n-1) | | average | std-dev (n-1) |
|---|---|---|---|---|---|
| go | 11.0401 | 5.4040 | ijpeg | 13.6127 | 8.4577 |
| m88ksim | 10.7539 | 3.9189 | perl | 13.0473 | 7.0491 |
| gcc | 11.4777 | 11.1734 | vortex | 24.0236 | 10.1844 |
| compress | 12.6343 | 11.3130 | tomcatv | 11.8892 | 6.8793 |
| xlisp | 12.7525 | 7.2824 | | | |

Table 2.5: Branch resolution delays for conditional branches

should be independent of the predictor organization, except as differences in prediction change the instruction flow through the processor.

Such long resolution times typically mean the processor has multiple branches in flight, in different stages of the processor. This requires a structure in which shadow register maps, return-address-stack patch state, and if applicable, branch-history fixup state are stored. The structure has a fixed depth—20 entries in the Alpha 21264 [49], for example—beyond which fetch stalls. For simplicity, simulations assume unlimited shadow-state capacity except where noted.

## 2.3 Selecting a Single, Representative Sample for Efficient Simulation

The SPECint programs typically run for billions of instructions using reference inputs. Even smaller inputs typically run for hundreds of millions. But the fastest detailed microarchitecture simulators still take about an hour per 100 million instructions simulated on an UltraSPARC II or Pentium Pro. Simulating to completion is therefore usually too expensive, especially for studies like this paper, which need hundreds of separate simulation runs.

Using very small inputs and scaling hardware structures accordingly is one possible solution. Scaling is risky, however: appropriate scaling factors are not always evident, and one must be careful to scale all relevant structures appropriately. Using smaller inputs may also change the relationship among various factors: for example, if various loops now iterate less often, branch prediction behavior may change, inherent ILP may change, and so forth.

This work describes a simple alternative: selecting a small, representative simulation window from a full-length run with the reference input.

Many researchers use a single, small, simulation window because it makes simulation so much simpler. Indeed, this research generally simulates just 50 M instructions for each program. This section argues that a small simulation window like this, if chosen carefully, can be reasonably representative of general program behavior. This makes it possible to simulate many different configurations or many benchmarks in a short period of time.

### 2.3.1 Related Work

Sampling schemes are widely used in architecture studies and several pieces of prior work have investigated sampling methodologies, particularly related to cache memory simulations. For example, Laha *et al.* [54] studied the accuracy of memory reference trace sampling using caches that were 128 KB and smaller. Their study concluded that sampling allows accurate estimates of cache miss rates, but their results were presented for fairly unaggressive sampling techniques that simulate 60% of all memory references.

As the fraction of references or instructions modeled becomes smaller, the question of how to "prime" the cache—how to deal with the unknown cache state at the beginning of each sample—becomes important. This work takes a brute-force approach and simply simulates all instructions preceding the desired sample, just at a lower level of detail. Only the model's caches, branch predictor, and architectural state are updated. Other work has studied analytic models for estimating cache miss rates during the unprimed portion of the sample [48, 112], or described means for bounding errors by adjusting simulation lengths [62]. Iyengar and Trevillyan have derived the *R-metric* for measuring the representativeness of a trace [37], and they generate traces by scaling basic-block transition counts and adjusting selected instructions to optimize the R-metric. Their technique incorporates cache and TLB behavior as well as branch prediction behavior, but because it uses traces, important mis-speculation effects may be omitted.

A shorter version of this chapter's discussion appears in [84].

### 2.3.2 Using Interval Traces to Select the Simulation Window

To accurately choose the simulation window, I have first measured *interval* branch misprediction rates for each of the benchmarks: *i.e.*, the misprediction rate computed separately over each million-instruction interval in the program. This helps expose small but representative segments of the programs, while smoothing out the finest-grain variations in behavior. To illustrate this, Figures 2.1–2.3 show the interval-branch-misprediction traces for the benchmarks, on which a pair of vertical lines delineates the 50 M instruction segment chosen as the simulation window. Note that *m88ksim*'s trace resembles *perl*'s in terms of having a short initial phase and a fairly flat trace afterwards, and *compress, vortex*, and *tomcatv* resemble *ijpeg* in terms of having clearly distinct, repeating phases. *Compress*'s phases, however, correspond to successive compress and decompress phases which are artifacts of the benchmark version. *Compress* also has a dramatic startup phase of about 1.7 billion instructions during which it generates the data to be compressed and decompressed. This is also an artifact of the benchmark version, but the benchmark versions are the ones many architects use for their simulations. In fact, the misprediction rate during the startup phase is inordinately high—14.5% compared to a maximum of 11% for the rest of the program—and this creates a risk of substantially unrepresentative results if simulations include too much of this segment of *compress*'s execution relative to

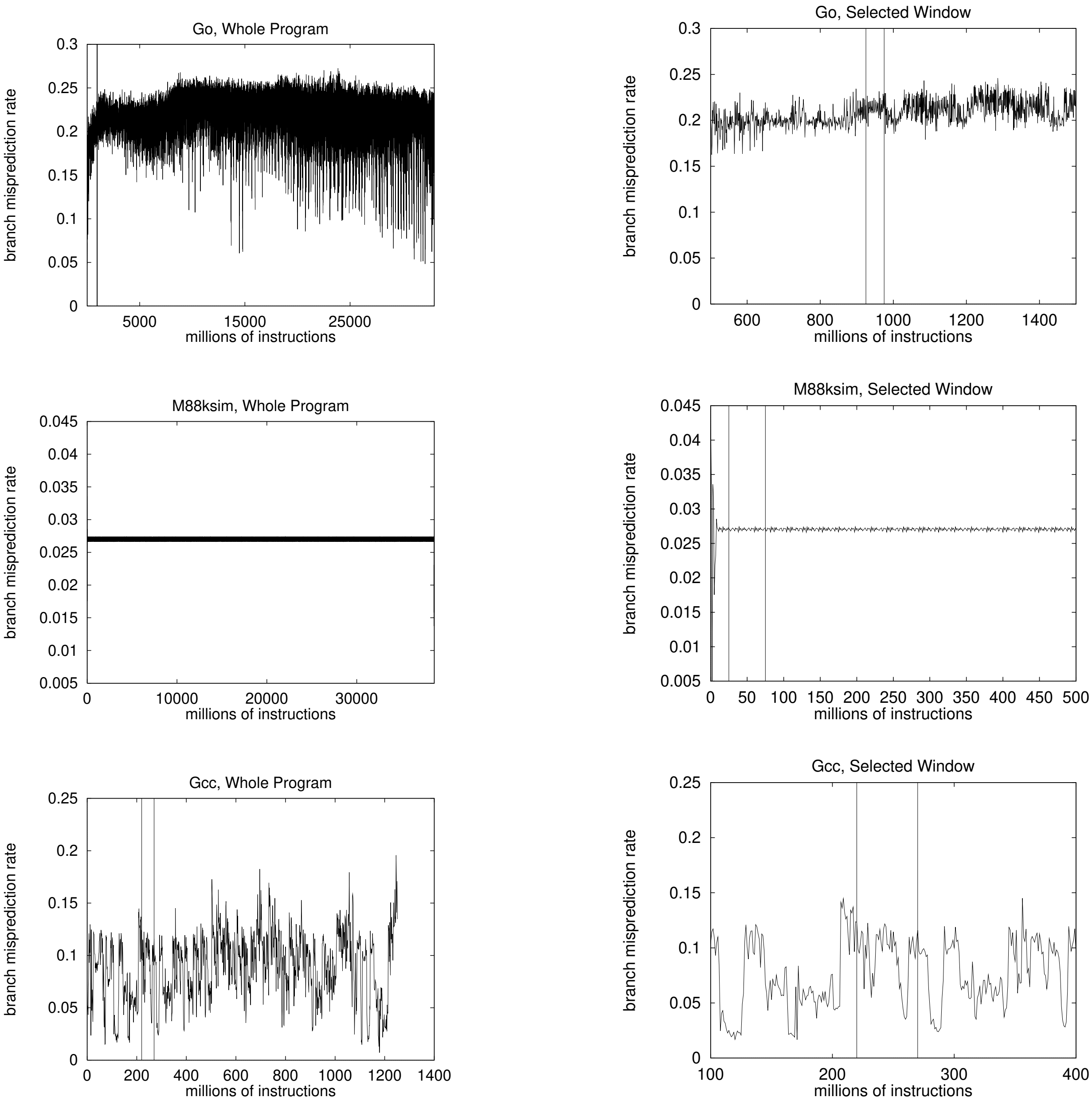


Figure 2.1: Interval miss rates for go, m88ksim, and gcc.
The left-hand chart shows a trace from the entire program (using only the inputs described in Section 2.2), and the right-hand chart shows a small segment of the trace. In both charts, two vertical lines indicate the chosen simulation window (but in the left-hand charts, these vertical lines are sometimes too close to the x-axis.

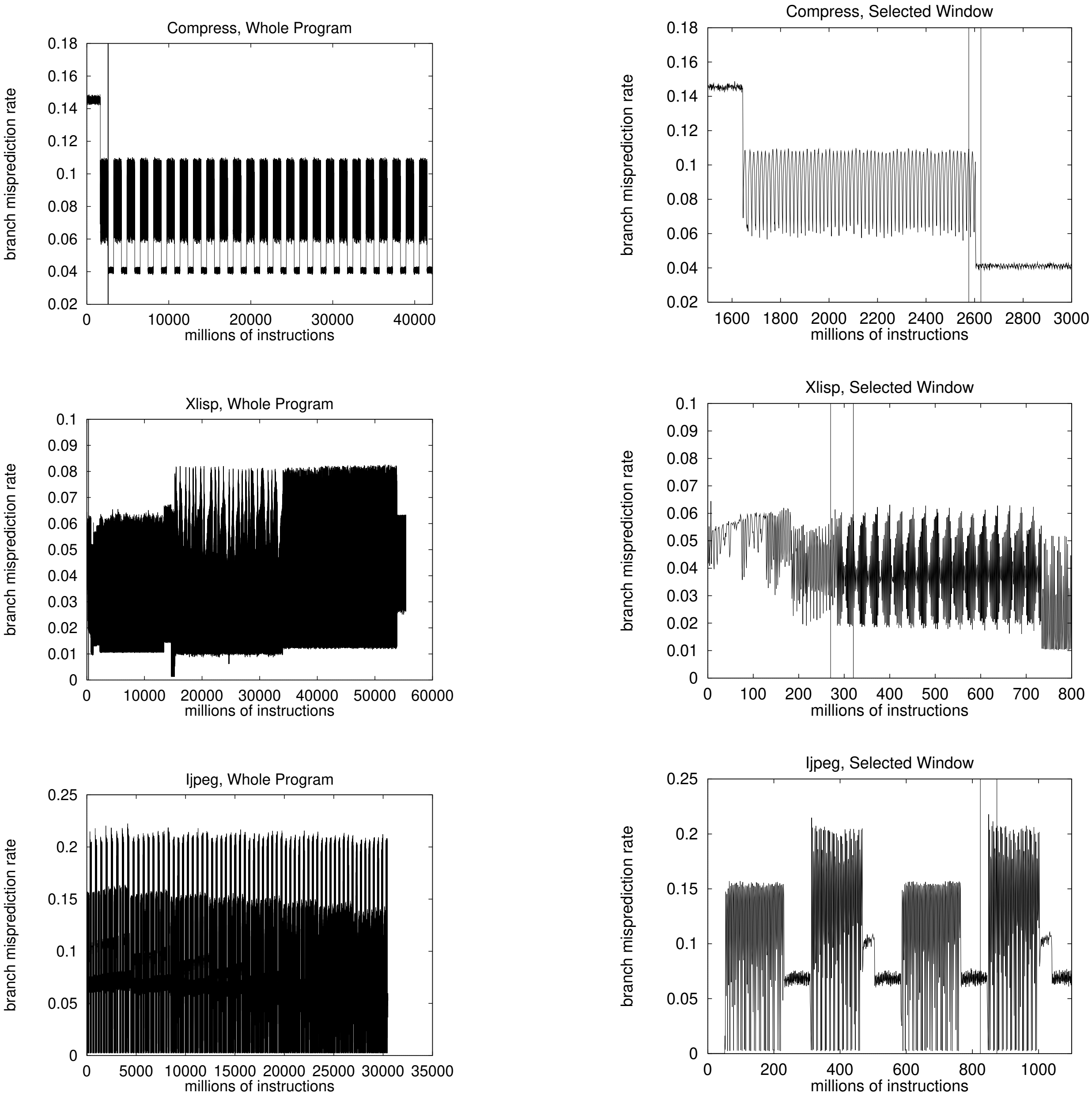


Figure 2.2: Interval miss rates for compress, xlisp, and ijpeg.

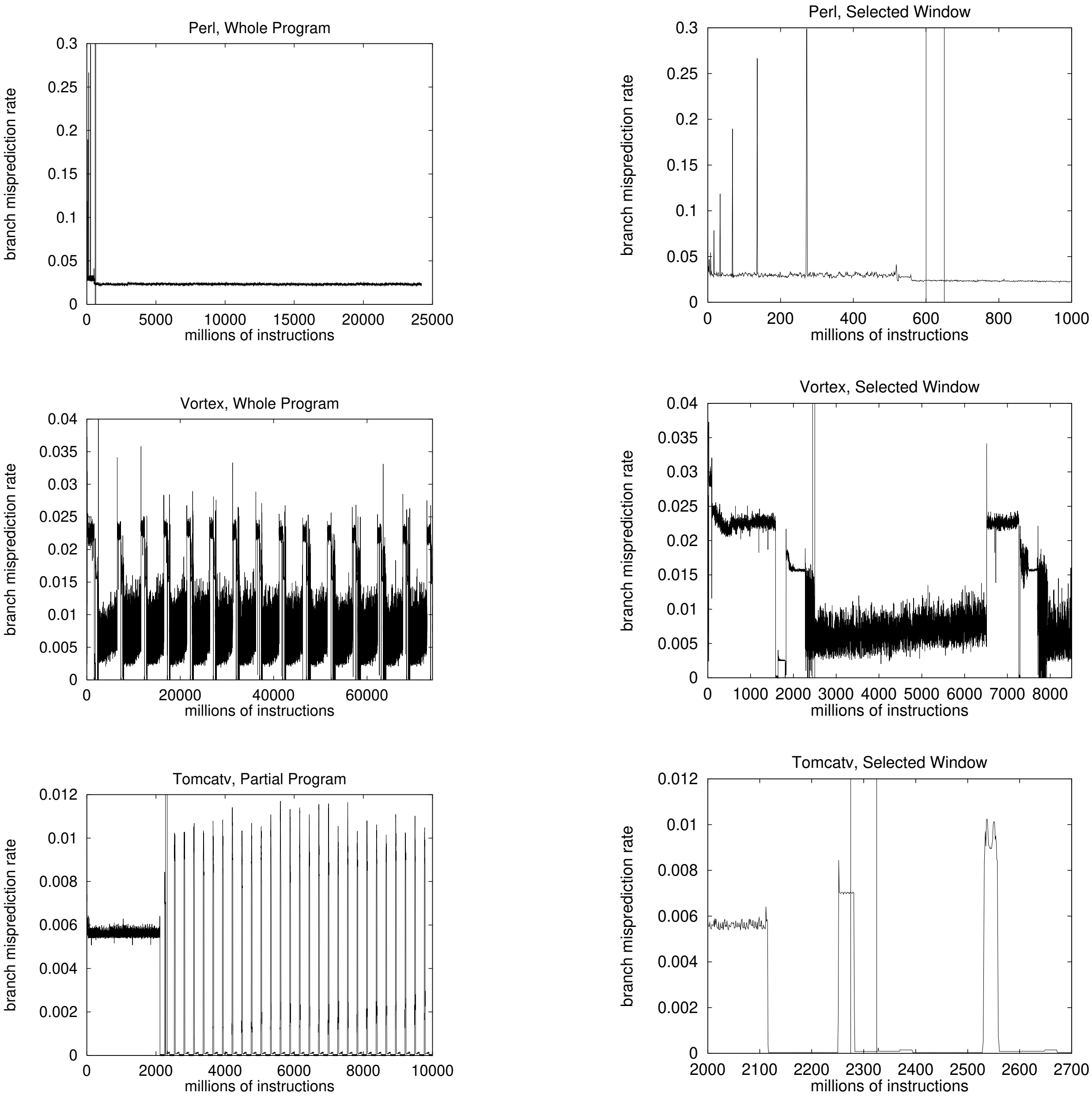


Figure 2.3: Interval miss rates for perl, vortex, and tomcatv.
Tomcatv with reference data runs for so long that a chart for the entire program does not show useful detail. Instead, the left-hand chart shows a trace for the first 10 billion instructions.

actual compress/decompress behavior.

Interval traces were also obtained for data and instruction cache miss rates to ensure that the chosen segment was suitable with respect to cache as well as branch-prediciton behavior. The different types of traces show similar qualitative behavior, with major shifts occurring at the same point in each type of trace for a particular benchmark. Cache traces are omitted here but appear in [89].

### 2.3.3 The Impact of Programs' Startup Behavior

As mentioned with regard to *compress*, most programs show some sort of startup behavior. In several cases the corresponding misprediction rate differs markedly from that during later phases of the program: *go, compress, perl*, and *tomcatv* are examples. If this initial behavior represents too large a fraction of the simulation, consequent results are unreliable. In particular, a number of studies simulate the first 50–100 M instructions from each benchmark, which often includes only startup behavior.

To show how significantly results can change when the 50 M instruction simulation window is dominated by startup behavior, Figure 2.4 plots three-dimensional graphs of IPC vs. RUU size vs. data cache size for *go, gcc, compress*, and *perl*, for both regular and 100% branch prediction. The left two columns' worth of graphs show data for baseline branch prediction, and the right two columns show data for 100%-direction prediction. In each case, an accurate 50 M instruction simulation window that avoids startup behavior (using the warmup times given in Table 2.2) is compared with a window taken from the beginning of the program's execution (using a warmup period of just one million instructions to prime processor structures). Within each graph, the vertical, or z-axis presents performance in terms of average instructions per cycle, or IPC. Each bar represents a single simulation with one combination of L1

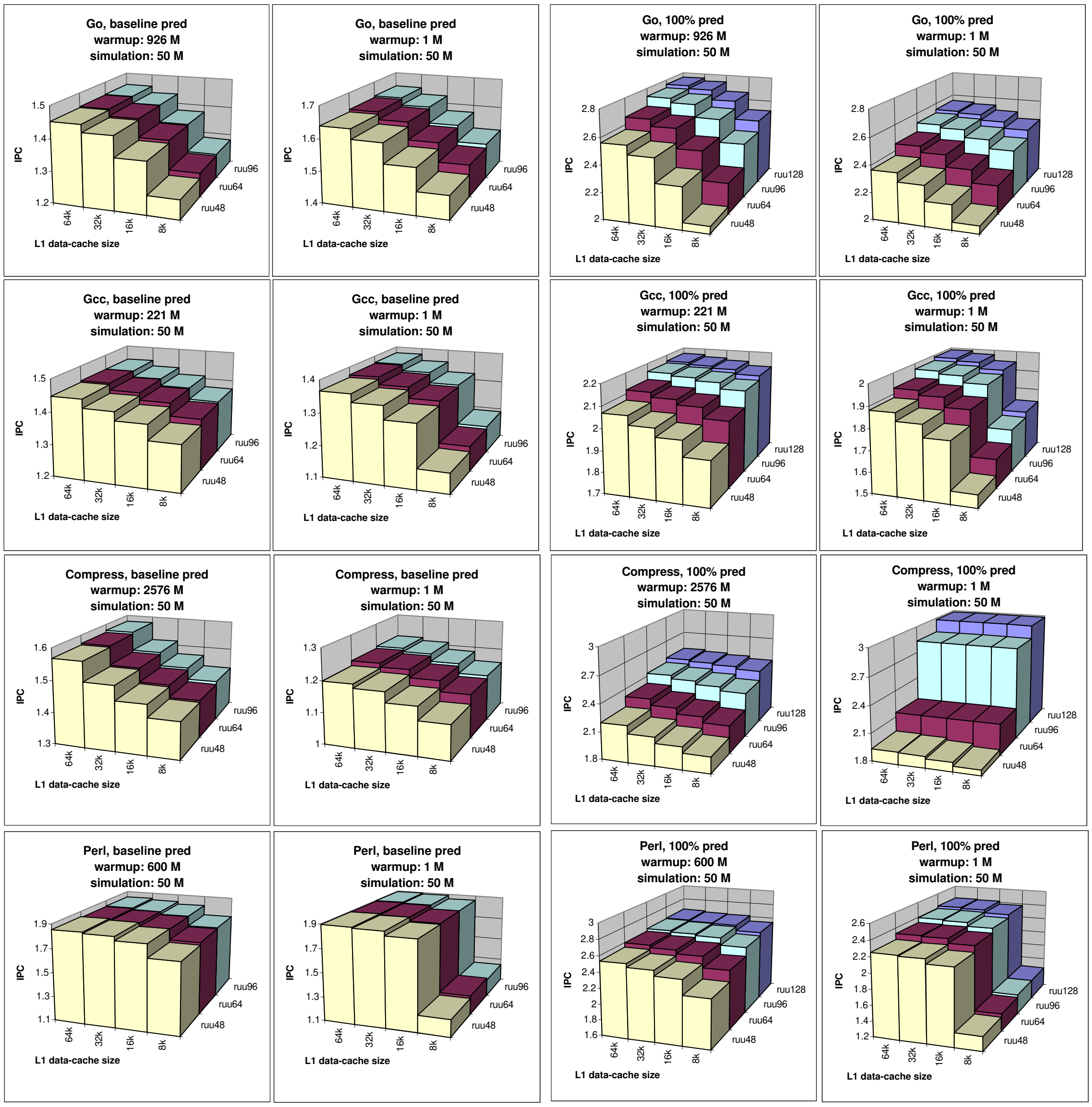


Figure 2.4: Comparison of IPC/D-cache/RUU surfaces for 50 M-instruction simulations and no simulator warmup vs the chosen warmup interval.

cache size (plotted along the left-to-right horizontal axis) and RUU size (plotted along the front-to-back horizontal axis), and all other baseline parameters held constant.

For all benchmarks, behavior between the two cases—between the first and second graphs, and between the third and fourth graphs—differs markedly, even though the one million instructions of warmup eliminate the most egregious startup effects. For *go, compress*, and *perl*, the initial phase has substantially different branch prediction behavior, so such differences come as no surprise. *Gcc*, on the other hand, has no clear initial phase, but second-level cache misses contribute significantly, because 1 M is too short a warmup for big L2 caches. Of course, as simulations run longer, distortionary effects from startup behavior have less impact on overall results. On the other hand, the startup phase can be extremely long: *e.g.*, 1.5 B instructions for *compress*, and over 2 B for *vortex* and many floating-point programs—prohibitively long for full-detail simulation.

The simulation window should therefore be placed after any such initial phases. Fortunately, this can be done fairly quickly using fast-mode simulation. SimpleScalar has recently begun offering a checkpointing facility that removes even the need for fast-mode simulation [4]. Unless the initial behavior dominates the rest of the program's execution, omitting it still gives reasonable results.

### 2.3.4 Evaluating the Effectiveness of Small Simulation Windows

A 50 M instruction simulation window reliably gives representative results for SPECint programs, if the window is carefully chosen and is taken from later in the program's execution. As evidence for this, Figures 2.5 and 2.6 give further 3D graphs comparing the chosen 50 M window to a 250 M instruction simulation window (I have

500 M-instruction data for *go*, and present that instead). Again, compare the first graph to the second, and the third graph to the fourth. Particular IPC values change slightly, but the IPC-cache-RUU surfaces remain similar. The one slight exception is *gcc*, for which the step from an 8 K to a 16 K cache is not as pronounced for 50 M as for 250 M. Data with a 100 M window does match the 250 M data. The discrepancy is due to L2 cache misses: with a perfect L2, the 50 M and 250 M data match closely. *Ijpeg* is also sensitive to L2 misses, because the L2 miss behavior does not settle down until several execution phases have passed. But consulting the L2 interval-miss-rate plot when choosing a simulation window produced the 50 M instruction segment used here, and it does capture representative behavior.

Gathering the data for Figures 2.5 and 2.6 forced me to refine my initial choice of window in the case of *ijpeg* and *gcc* (by consulting the L2 trace). Choosing a representative 50 M-window window using the interval traces is a good heuristic, but verifying it by comparing 50 M-instruction plots of IPC vs. RUU size vs. D-cache size against 250 M-instruction plots provides even more reliable simulation windows.

Note that for programs with distinct phases of execution, like *compress* or *ijpeg*, the window has been placed so that it straddles the two phases (fortunately, these SPECint programs have at most two phases). Space considerations prevent us from presenting further 3D graphs, but 50 M-samples taken from just a single phase do not match behavior from the other phase. Instead, as Figures 2.5 and 2.6 show, a straddling 50 M-window captures behavior that corresponds well to a much larger 250 M straddling window that includes a large portion of each phase. The smaller window therefore seems able to provide representative behavior even for programs like these that have two phases. Recall, however, that the phases in *compress* are an artifact of the SPEC benchmark version of this program. The different phases correspond to alternating passes of compressing and decompressing the same data.

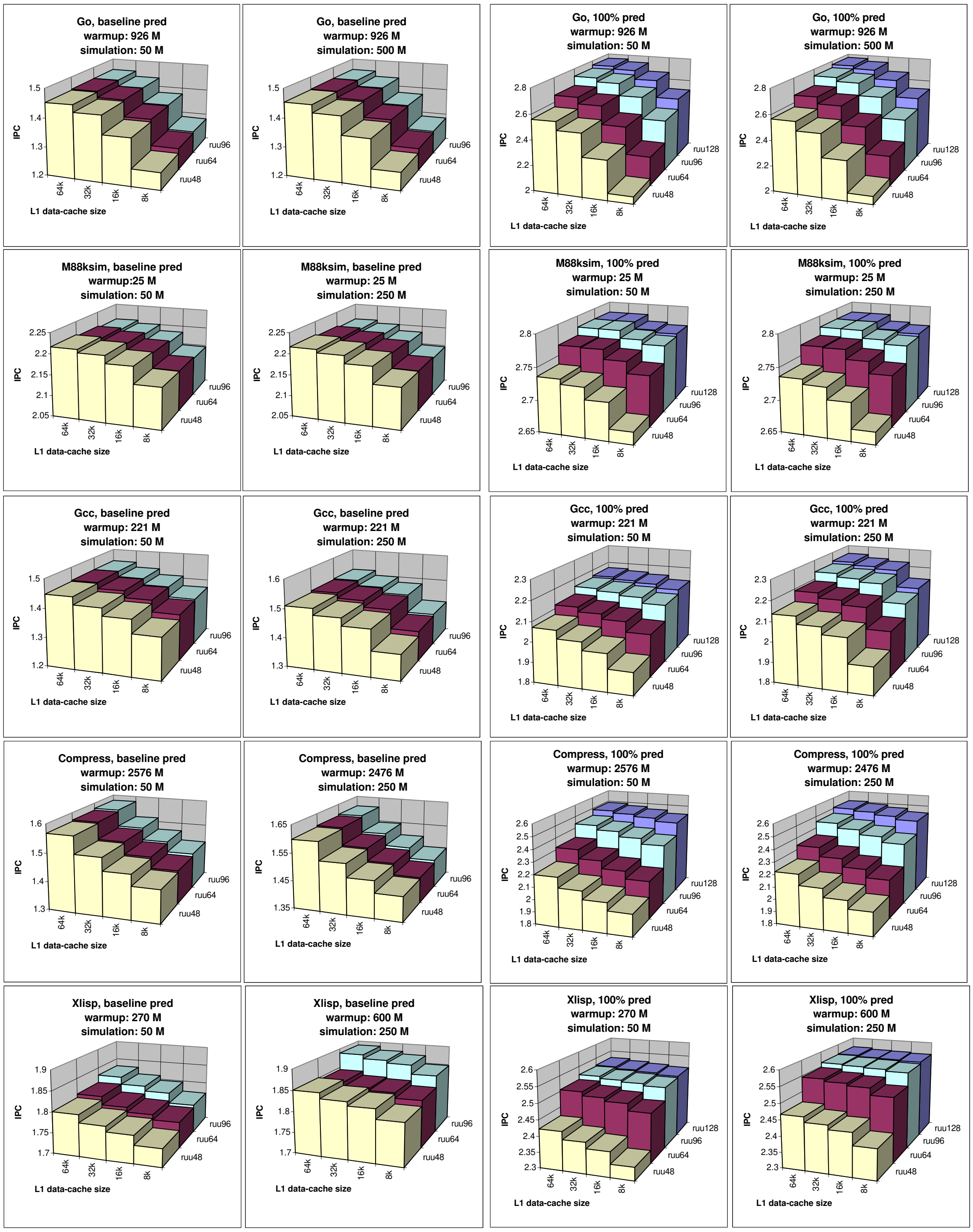


Figure 2.5: Comparison of IPC/D-cache/RUU surfaces for 50 M-instruction simulations vs. 250 M-instruction simulations.

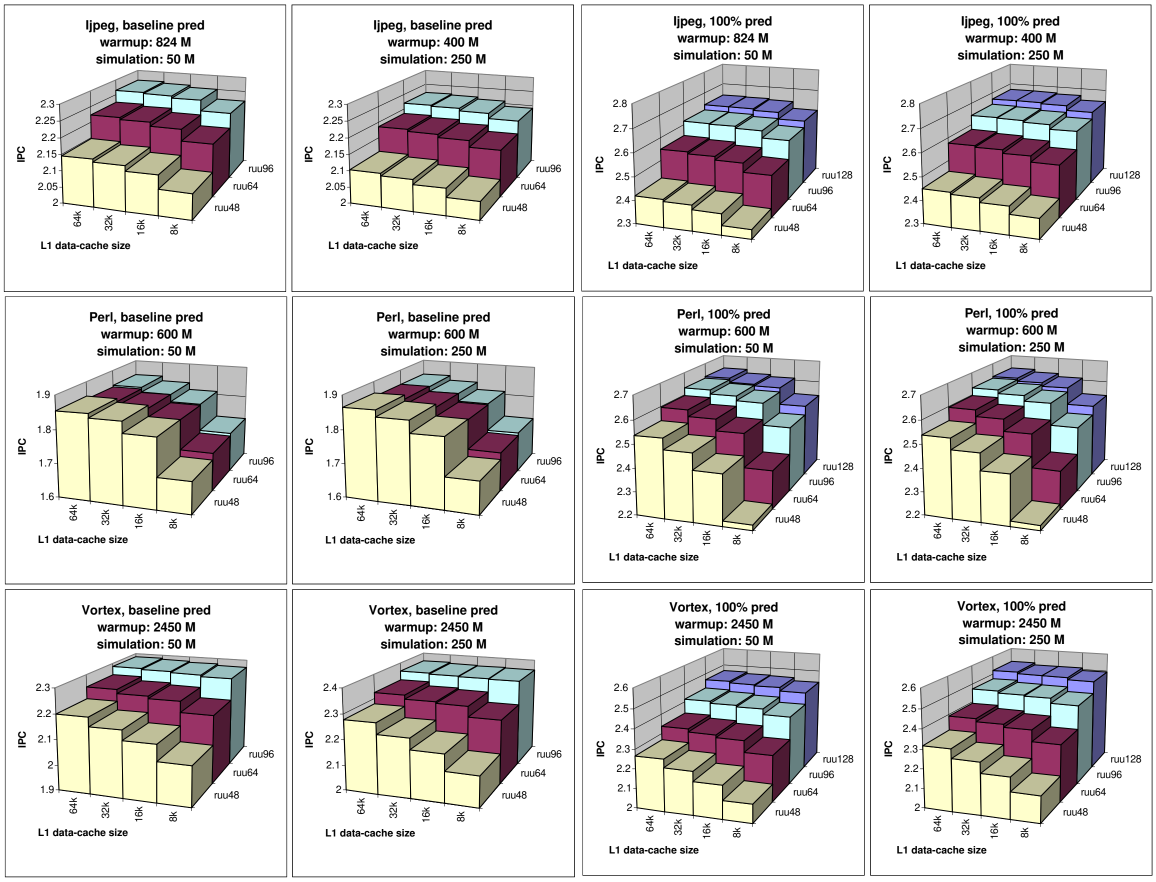


Figure 2.6: Comparison of IPC/D-cache/RUU surfaces for 50 M-instruction simulations vs. 250 M-instruction simulations.

### 2.3.5 Summary: Choosing a Simulation Window

In summary, this section has described a technique for choosing a short simulation window that is representative of the program's overall execution behavior. This permits simulations to use realistic, large inputs, while avoiding long simulation times. The steps involved are:

1. Collect interval traces of branch misprediction rates and cache miss rates, including L2 miss rates. This can be done with instruction-level simulation.

2. Identify a candidate simulation window of the desired length (*e.g.*, 50 million instructions). A good candidate should

   - Avoid program startup behavior
   - Capture representative branch prediction and cache behavior.
   - Capture representative behavior for any other aspect of importance. For example, when studying the design of the floating-point unit, the distribution of floating-point operations or even their operands may need to be examined.

3. Compare the IPC-RUU-data cache size surfaces for the candidate simulation window and for a much longer simulation window (5–10 times as long). Ideally, the longer simulation window would consist of the program's entire execution, but for many programs this is infeasible even if only done once. Note that cycle-level simulations are required to produce these 3D graphs.

   When studying other aspects of processor design, like the floating-point unit, additional 3D graphs that test sensitivity to the feature in question should be generated. Satisfactory IPC-RUU-data cache surfaces are still necessary, but when exploring other design issues, not sufficient.

4. If the 3D surfaces match reasonably, a suitable simulation window has been found. If the surfaces do not match, return to step 2.

The drawback to this approach is that it must be done by hand, and is labor-intensive. In practice, however, the necessary experiments are simple, and they only need to be done once for each program to be added to the benchmark set. The advantage to this approach is that it allows short simulation times. The IPC-RUU-cache size surfaces are sufficiently sensitive to provide reasonable confidence in the quality of the chosen simulation window. As a result, this simulation methodology avoids the unrepresentative program behavior that plagues many simulation-based studies, while at the same time avoiding long simulation times.

# Chapter 3

# The Need for Better Branch Prediction

## 3.1 Introduction

This chapter shows the importance of accurate branch prediction, and explores how branch prediction affects other structures in the processor. Many microarchitecture studies examine a single structure, like the branch predictor, the cache, or the instruction window. When viewing these structures individually, increasing their size generally increases performance, although points of diminishing returns exist. Yet die-size constraints make *tradeoffs* among these components necessary. The data presented here show how branch prediction, instruction window size, and cache size trade off against each other. In the process, this chapter shows how strongly the branch predictor's accuracy affects both overall performance as well as the behavior of other processor structures, especially the instruction window.

Unfortunately, the behavior of these structures is related. Changes to one cause different behavior in the others. With a certain branch predictor, for example, adding data cache may be more beneficial than increasing instruction window size. More is better—but a small improvement to the branch predictor might be better yet, possibly shifting the tradeoffs so that adding instruction window entries becomes more beneficial than adding cache. A branch prediction study that models unrealistic caches can under- or over-state the benefits of better prediction. If the caches are too small, for example, cache misses may mask the effects of branch mispredictions. Modeling caches that are perfect can be risky as well: some benchmarks stream data through even the largest caches, so omitting this significant contribution to execution time can lead to overstated benefits. Modeling perfect branch prediction is similarly risky for other types of studies. Making these problems worse is the fact that out-of-order issue and decoupling buffers cause latencies to interact in sometimes subtle ways. This makes the relative importance and inter-relationships of the major parameters more complex, and also makes more difficult the task of finding sensible configurations for a simulation-based study.

While such pitfalls may strike many readers as obvious, methodological mistakes like these are still common. Yet running pilot experiments to insure against such potential pitfalls and to cull the design space is expensive. A further problem is that detailed simulation is slow, and generally prohibits executing realistic benchmarks to completion without sampling or reducing input sizes. In conjunction with the previous chapter's description of how to find a short but representative simulation window, this chapter addresses these problems. It characterizes benchmark performance in terms of cache size and instruction window size to help cull the design space that researchers need to explore. For example, several SPECint95 benchmarks [98] fit in 8 K instruction caches; while the small instruction cache footprint of SPEC benchmarks is a common complaint, this chapter provides data confirming this. The chapter also discuss tradeoffs among cache size, instruction window size, and branch prediction accuracy. When not taken into account, these tradeoffs can skew simulation results in damaging ways. For example, assuming perfect branch prediction makes large (*e.g.*, 128-entry) instruction window sizes look artificially beneficial.

The most important result from these studies is the demonstration of how critical branch prediction accuracy is. Branch mispredictions directly stall the fetch engine (in reality, the fetch engine continues fetching on the mispredicted path, but these wrong-path instructions are squashed once the misprediction is detected). This means that during a misprediction, no useful instructions are fetched. This misprediction latency can be quite long; the minimum misprediction latency in the recently released Compaq Alpha 21264, for example, is seven cycles [49], and the minimum latency in the Intel Pentium Pro is eleven cycles [31]. Average latencies are often longer in out-of-order processors, because the branch instructions may spend an arbitrarily long time in the instruction window before they resolve. Average latencies range from 10–20 cycles.

In addition to disrupting the processor pipeline, branch mispredictions disrupt the efficacy of other processor structures. The instruction window provides the most dramatic example. As shown in the next section, even with prediction accuracies above 90%, mispredictions still occur so often that the instruction window can only rarely accumulate a pool of more than 48 to 64 entries. Increasing instruction window size beyond this level therefore provides no added benefit. On the other hand, if branch prediction accuracy can be boosted 97 to 100%, the story is quite different, and much larger instruction windows become beneficial. Many programs have substantial instruction-level parallelism (ILP) [106], but branch mispredictions are a critical impediment to exposing it.

### 3.1.1 Contributions

To show the tradeoffs among branch prediction, instruction window size, data cache size, and instruction cache size, this chapter shows performance as a function of these parameters for each SPECint95 benchmark. To explore the impact of branch prediction accuracy, changes in the instruction window size/data-cache size tradeoff are plotted for a range of branch prediction accuracies. Comparing data for an ideal branch predictor against that for a baseline, hybrid branch predictor like the Alpha 21264's [32, 49] shows how much branch mispredictions limit performance; more importantly, it illustrates the circumstances under which simulating ideal prediction skews the results. Note that this work also appeared in [84].

Not only is it important to understand how branch prediction limits the performance of other processor structures, it is crucial to understand the interplay of all these parameters, and this chapter serves as a reference aid for this. I also hope that the comprehensive data presented will help researchers reduce their simulation requirements in various ways by describing a methodology for culling the simulation space and providing reference data for that purpose. This work focuses on the SPEC suite of benchmarks both because they provide consistency and comparability across studies, and because there are few other agreed-upon benchmarks that are portable and publicly available with source code. As mentioned earlier, the SPEC integer programs are of primary interest because because the SPEC floating-point programs have near-perfect branch prediction.

In particular, the results show not only that branch mispredictions limit the benefits of larger instruction windows, but that better branch prediction and better instruction cache behavior have synergistic effects, and that the benefits of larger instruction windows and larger data caches trade off and have overlapping effects.

### 3.1.2 Simulator Details

This chapter uses HydraScalar for all its measurements, and simulates each benchmark for 50 million instructions, chosen according to the method described in the previous chapter. The simulator is also configured as described in the previous chapter.

The branch predictor used for "baseline" branch-prediction results is a McFarling-style hybrid branch predictor [64] that combines two two-level prediction mechanisms [115] with a selector that chooses between them. The two components are a GAg (global history) predictor with a 12-bit history, a 4K-entry PHT, and two-bit saturating counters in the PHT; and a PAg (local history) predictor using 10-bit histories, a 1K-entry PHT, and three-bit counters. For each predictor, the selector chooses the component most likely to be correct by consulting its own 4K-entry table of saturating two-bit counters, also indexed by 12 bits of global history [11]. This branch predictor models the hybrid predictor used by the Alpha 21264. The only difference is that the model used here does not update the global history speculatively, while the 21264 does. For more information on speculative update, consult Chapter 5.

Two varieties of perfect branch prediction are also modeled in this chapter. The first, called *100%-direction*, correctly predicts the direction of every conditional branch, but does not prevent any BTB misses. The second, *oracle prediction*, correctly predicts every branch and destination, regardless of type.

In the rest of this chapter, Section 3.2 examines the relationship between performance and instruction window size, and the impact of branch mispredictions on this relationship. Section 3.3 examines the relationship between performance and both instruction and data cache size. Section 3.4 discusses branch prediction effects in more detail. Section 3.5 summarizes related work, and Section 3.6 sums up the major results in this chapter. All the data in this chapter have been obtained using the HydraScalar configuration given in Table 2.1 in the previous chapter.

## 3.2 Instruction Window Size and Branch Prediction

Figure 3.1 presents, for each benchmark, a 3D graph showing instructions per cycle (IPC) as a function of both RUU (instruction window) size and first-level (L1) data cache size. These measurements have been obtained using the baseline hybrid predictor described above. As in the previous chapter, each bar represents a single simulation with one combination of L1 size (plotted along the left-to-right horizontal axis) and RUU size (plotted along the front-to-back horizontal axis), and all other baseline parameters held constant. These step sizes were chosen to yield a feasible number of simulations spanning a reasonable range of sizes. The IPC scale varies among benchmarks to best show the detail in each surface.

Figure 3.2 presents similar data, but using the perfect direction predictor (BTB misses are still possible). It also extends the RUU axis to 256 entries. Note that the IPC scale changes between Figures 3.1 and 3.2.

First, let us consider performance as a function of RUU size, holding cache size constant and using the baseline branch predictor. The integer benchmarks fall into two groups, determined by the point at which performance as a function of RUU size plateaus. This plateau is the point at which additional entries contain useful instructions so rarely that overall performance is barely affected. Most benchmarks show a plateau in performance at 48–64 RUU entries, regardless of cache size. These benchmarks all have branch prediction accuracy below 95%. *M88ksim* falls in this category as well; it achieves 95% accuracy and its performance has already plateaued

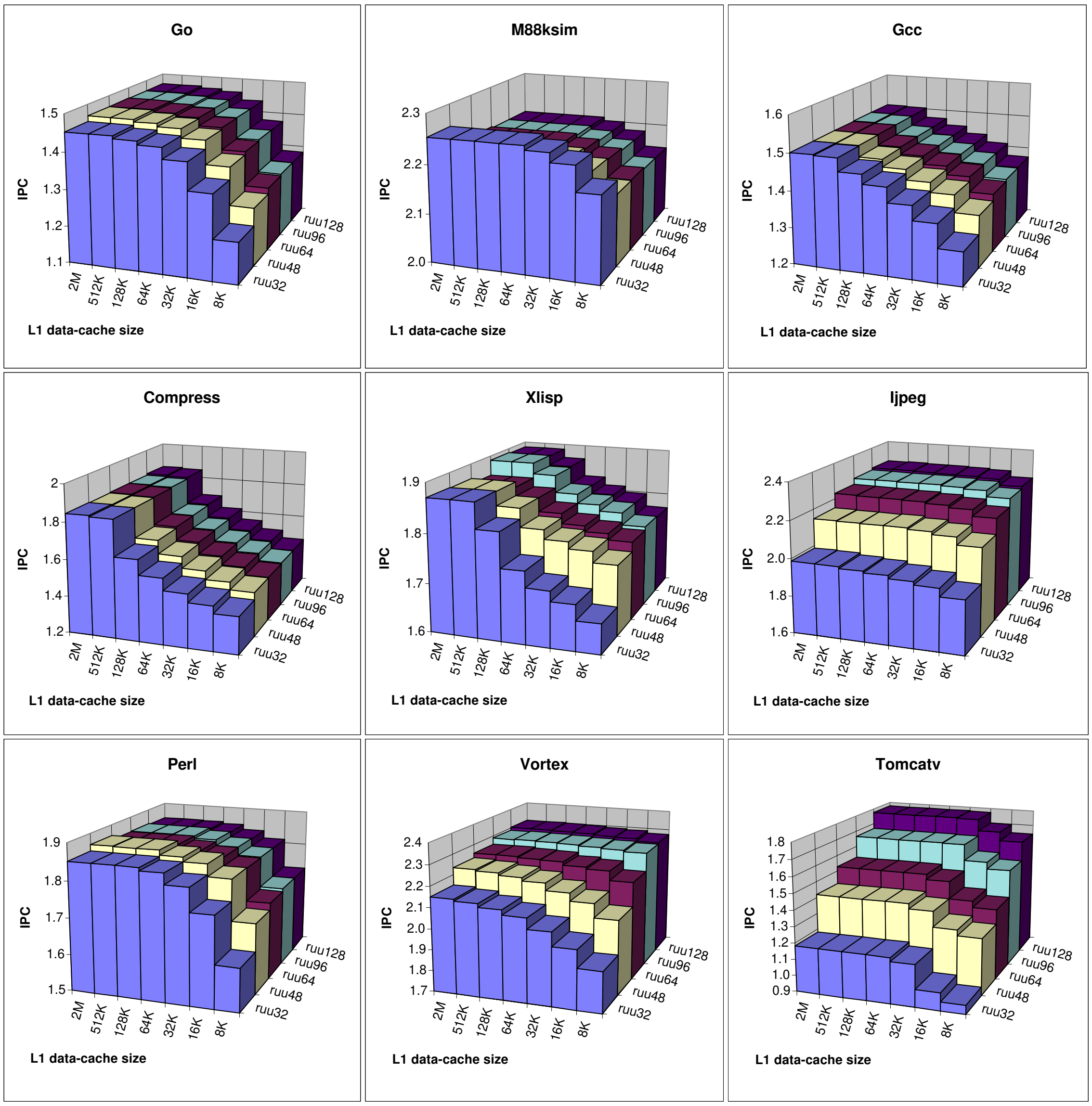


Figure 3.1: IPC as a function of data cache size and RUU size with the baseline, hybrid branch predictor.

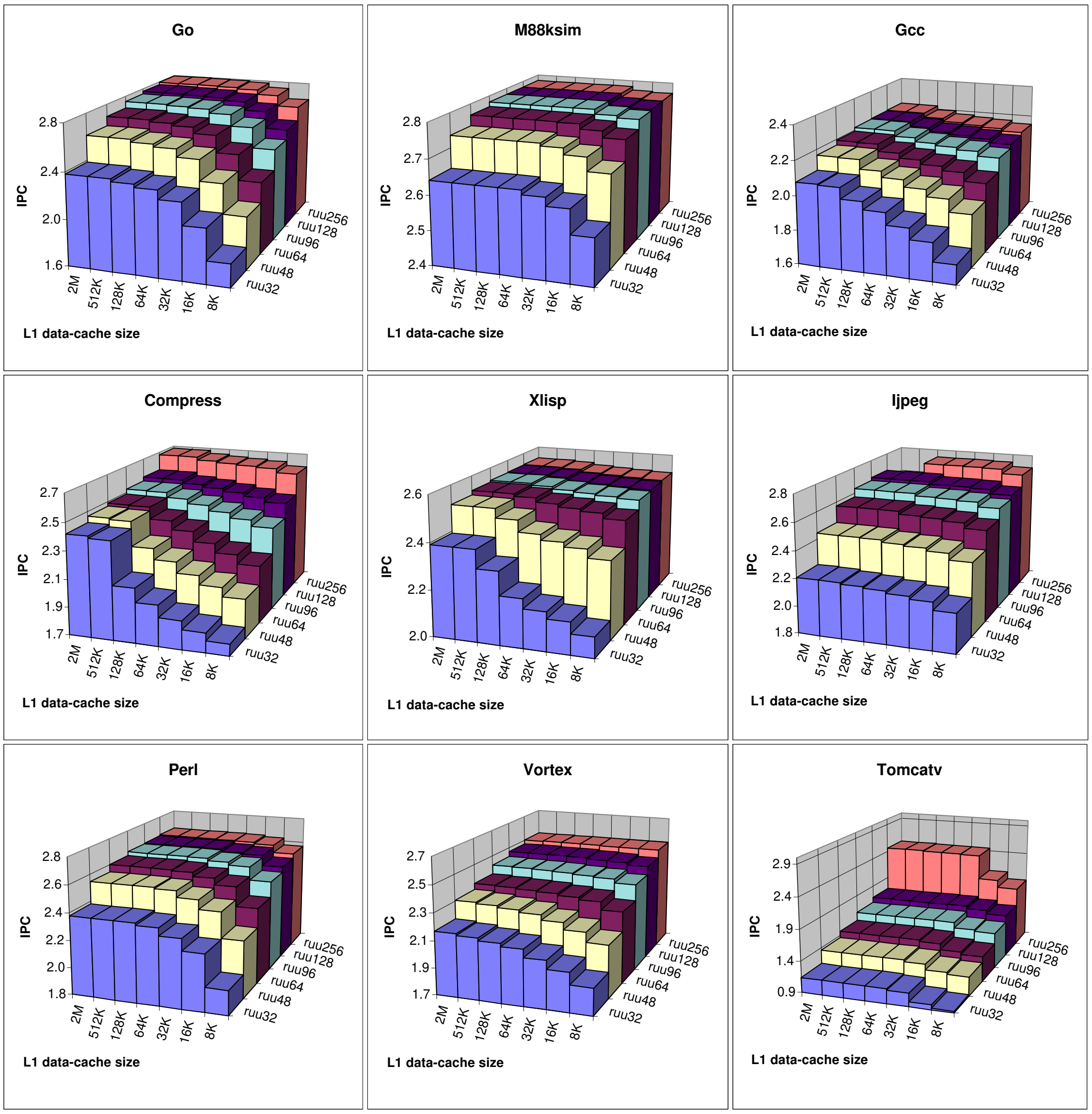


Figure 3.2: IPC as a function of data cache size and RUU size with the 100%-direction-accuracy predictor.
(Simulator difficulties prevented us from obtaining the the two ruu256 values for ijpeg with large caches.)

at 32 entries. (*M88ksim*'s decrease in IPC from 32 to 48 entries is discussed at the end of this section.) Note that contemporary processors also have RUU sizes in the 32–64 entry range. Thus, these results show that further increases in RUU sizes are unwarranted for many integer codes unless coupled with concomitant increases in branch prediction accuracy.

| | | | |
|---|---|---|---|
| go | 0.0275 | ijpeg | 0.0060 |
| m88ksim | 0.0085 | perl | 0.0083 |
| gcc | 0.0202 | vortex | 0.0026 |
| compress | 0.0198 | tomcatv | 0.0000 |
| xlisp | 0.0117 | | |

Table 3.1: Direction-mispredictions per instruction.

Two benchmarks, *vortex* and *ijpeg*, plateau later, at 96 entries. *Vortex* has a branch prediction accuracy of 98%, but *ijpeg*'s accuracy is only 88%. In fact, *ijpeg*, with one of the lowest accuracies, gets one of the largest gains among the SPECint programs from additional RUU. *Tomcatv*, with near-100% branch prediction accuracy, gets by far the most benefit from additional RUU; going from 32 to 128 RUU entries improves performance by 50% with a 64 K cache.

These data suggest that conditional-branch prediction accuracy is a helpful indicator of sensitivity to RUU size: except for *ijpeg*, the poorly-predicted programs don't benefit from larger RUUs. A more accurate indicator also factors in basic block size to obtain *mispredictions per instruction* [24]. Table 3.1 tabulates these rates. It shows that *ijpeg* benefits from further increments of RUU despite its low prediction rate because it has one of the lowest misprediction-per-instruction rates. Yet why do *m88ksim* and *perl* not benefit from deeper RUUs? They may have poor prediction accuracies, but they also have low values in Table 3.1. Poor prediction rates for *indirect* branches help explain the apparent discrepancies. Recall that, unlike a BTB miss for direct branches, which can be detected early, a BTB miss for indirect branches incurs a full misprediction penalty (seven cycles) in HydraScalar simulations. Table 2.3 in the previous chapter shows that even though these benchmarks have among the lowest direction-misprediction-per-instruction rates, they have among the worst accuracies for indirect branches: 25% for *m88ksim*, and 33% for *perl*. Indirect-branch accuracies also help explain why *vortex* plateaus earlier than *ijpeg*: although both programs have low direction-misprediction-per-instruction rates, *vortex* only predicts 77% of indirect branches correctly, while *ijpeg* predicts 98% correctly.

The tradeoffs change as prediction accuracy improves and bigger RUUs are more likely to be filled with correctly speculated instructions. At the extreme, 100% direction accuracy (Figure 3.2) most programs derive significant benefit from bigger RUUs, even out to 256 entries.

A high misprediction rate limits the benefit of additional RUU entries because mispredictions prevent additional RUU entries from helping performance in two major ways. Deeper RUU entries often go unoccupied—mispredictions flush instructions so often that the deepest entries rarely even become active—and if occupied, deeper RUU entries often contain mis-speculated instructions. Note that early branch predictor update (see Chapter 5) would boost the prediction accuracy and moderately decrease the impact of these effects, while longer misprediction-resolution times (*e.g.*, longer pipelines) would increase the impact.

Figure 3.3 illustrates the first effect for several representative benchmarks. The left-hand charts show, for a 128-entry RUU, the cycle-by-cycle distribution of RUU occupancies for the baseline branch predictor; the right-hand charts show the same data for 100%-direction prediction. An RUU entry is considered occupied in a cycle if it contains any instruction, whether already executed or not, and whether mis-speculated or not. The bars use the left y-axis, and show percentage of cycles that

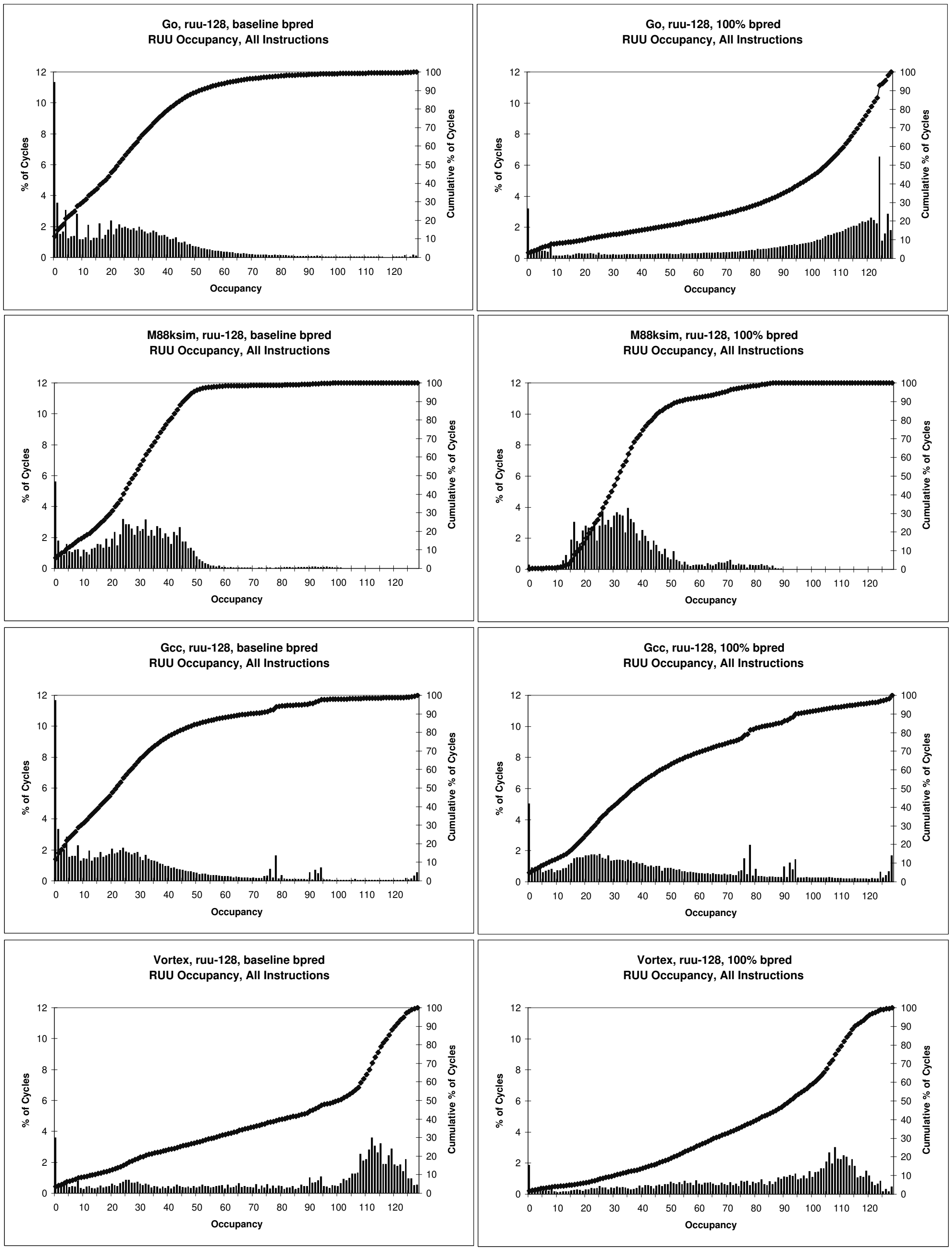


Figure 3.3: Percentage of cycles in which the RUU contains a particular number of instructions. Data appears for both the baseline hybrid scheme and for perfect branch prediction. The per-occupancy bars use the left-hand y-axis; the cumulative curves use the right-hand y-axis. Any instruction in the RUU, even a mis-speculated one or one waiting to retire, counts toward the occupancy.

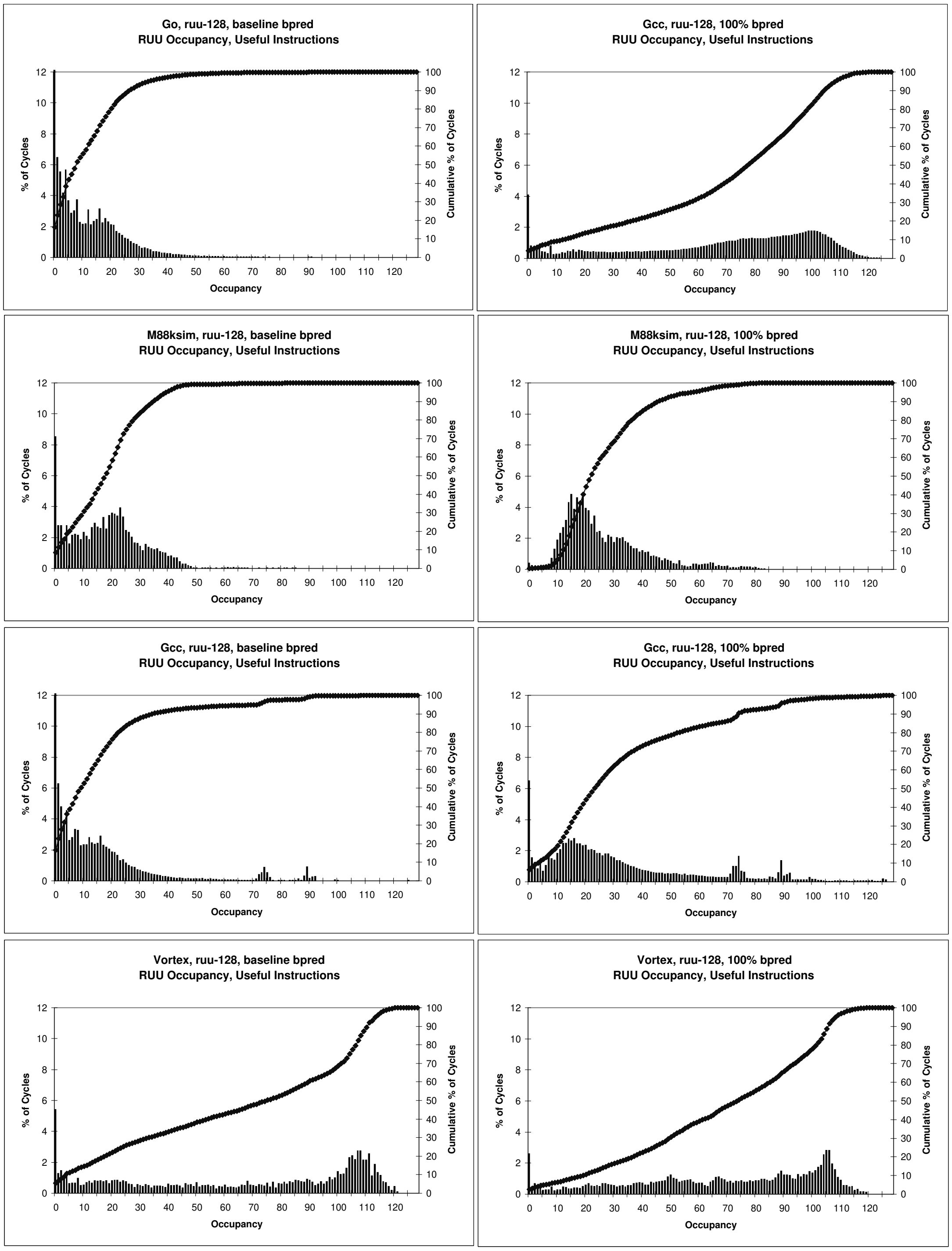


Figure 3.4: Percentage of cycles in which the RUU contains a particular number of correctly-speculated instructions, for both the baseline and for perfect branch prediction. Only instructions that have not yet issued are counted. The y-axis here has been truncated for go and gcc; for both, the 0 column has a value of 16.5%.

the RUU has a specific occupancy. The graphs show cumulative curves of the same data using the right y-axis.

For *gcc* with imperfect prediction and a 128-entry RUU, entries beyond the 64th are occupied only 11% of the time, and entries beyond the 96th only 2% of the time. Half of this large RUU is idle most of the time. In fact, although the data do not appear here, the deeper half of just a 48-entry RUU is idle more than 50% of the time. Deeper entries are used so infrequently that they can at best have a limited impact on performance. The low average occupancy results mainly from frequent squashing of instructions after a misprediction is identified. With perfect prediction, the picture looks quite different, as the right-hand side of Figure 3.3 shows. Even the deepest entries of a 128-entry RUU get regular use, and the RUU is often close to full.

The programs not shown have behavior resembling one of the programs in Figure 3.3. For *perl*, plots like those in Figure 3.3 closely resemble the plots for *go*. *Compress* and *xlisp* resemble *gcc*, and *ijpeg* resembles *vortex*.

With poor branch prediction, when a deeper RUU entry does become active, it often merely receives a mis-speculated instruction. These instructions can have helpful or harmful cache effects, but otherwise are useless and may cause contention for resources. Instructions which have completed execution and are waiting to retire also waste RUU entries. Figure 3.4 shows the cycle-by-cycle distribution of RUU occupancies for instructions that have been correctly speculated and have not yet executed. These correspond to issue-queue occupancy and are naturally lower than the overall occupancies in Figure 3.3. For imperfect prediction with *gcc*, useful occupancy exceeds 64 entries only 5.5% of the time, and exceeds 96 only 0.25% of the time. With 100%-direction prediction, on the other hand, useful occupancy for *gcc* exceeds 64 entries 15.5% of the time, and exceeds 96 entries 2.4% of the time. The change is more dramatic for other programs like *go*: with the baseline predictor, useful occupancy exceeds 64 entries not even 0.5% of the time, while with 100%-direction prediction, useful occupancy exceeds 64 entries 64% of the time.

Even if deep RUU entries receive a useful instruction, that instruction often waits for its operands. If the instruction waits so long that a smaller RUU could also have fetched it before the operands were ready, the earlier fetch and decode afforded by the larger RUU does not help. In fact, instructions almost never issue from beyond the 48th–64th entry, but recall that deeper entries rarely contain any useful unissued instructions except with perfect branch prediction.

RUU size tends to matter more with smaller first-level data caches than with large caches, because smaller caches provide more miss latencies that can be overlapped with other instructions from the RUU. This is evident in Figure 3.1 for *go, gcc, compress, xlisp, perl, vortex*, and to a lesser extent, *ijpeg*: the slope along the RUU axis (front-to-back) becomes shallower as the data cache becomes larger. Some instances do arise, however, in which bigger caches cause RUU size to matter more. Consider the step from a 64-entry to a 96-entry RUU for *xlisp*: with an 8 K cache, Figure 3.1 shows no difference, while with a 512 K cache, a small improvement appears.

Two more consequences of branch mispredictions might keep larger RUUs from helping. First, if a larger RUU causes branches to take longer to commit (because the branches are fetched earlier), branch prediction accuracy may decline as a result. The branch predictor is updated when branches commit; between the time of a branch's fetch and its commit, subsequent branches will see slightly stale branch predictor state. Average branch-resolution time for *m88ksim* increases by 17% when moving from a 32 to a 48-entry RUU, and its conditional-branch accuracy in turn falls by 1%. *M88ksim's* better performance for a 32-entry RUU than for larger RUUs is most likely due to this effect. The same effect may explain, in Figure 3.1, the small decline for *perl* for 64-entry and larger RUUs, and the small decline for *xlisp* for a 64-entry

RUU with large caches.

Second, mis-speculated instructions can also have cache effects. These can hurt performance by creating contention or by causing extra instruction or data cache misses; on the other hand, and they can have a prefetching effect for caches [71]. The performance impact of these effects is less than 1% for all the benchmarks.

## 3.3 Sensitivity to L1 Cache Size

### 3.3.1 Instruction Cache

Figures 3.5 and 3.6 show further 3D graphs of IPC as a function of cache size and RUU size; here, however, instruction cache (I-cache) size is explored instead of data cache (D-cache) size. Some of the benchmarks—*compress, ijpeg*, and *tomcatv*—fit even in an 8 K I-cache, and plots for these can be safely omitted. But for others, Figure 3.5 shows that instruction cache size profoundly affects performance. With the baseline branch predictor, increasing cache size from 8 K to 16 K improves performance by 20 to 40%—much more for *m88ksim*—and increasing cache size from 8 K to 128 K nearly doubles performance for *gcc, perl*, and *vortex*. So although the SPEC benchmarks are notorious for their small cache footprints, several still benefit from larger I-caches than those found in processors today.

I-cache size has even more impact with 100% direction-accuracy branch prediction, as seen in Figure 3.6. This is partly because perfect prediction eliminates the prefetching done by mis-speculated paths, but this mechanism only affects IPC by about 1%. Most of the difference is due simply to eliminating time wasted on mis-predictions.

Tables 3.2 and 3.3 present instruction cache miss ratios for the baseline and 100%

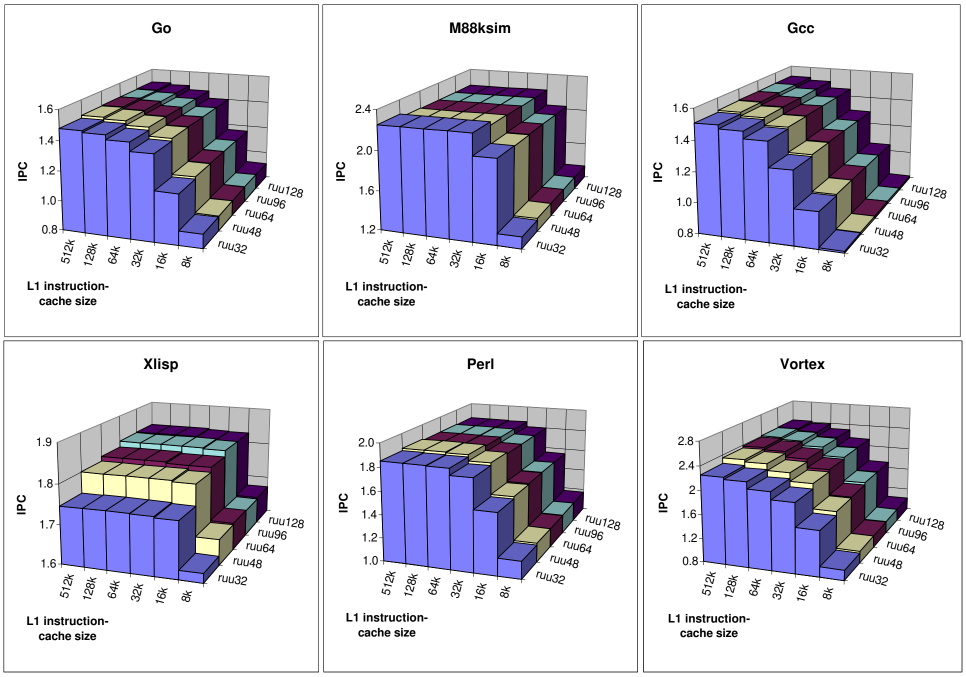


Figure 3.5: IPC as a function of instruction cache size and RUU size with the baseline, hybrid branch predictor.

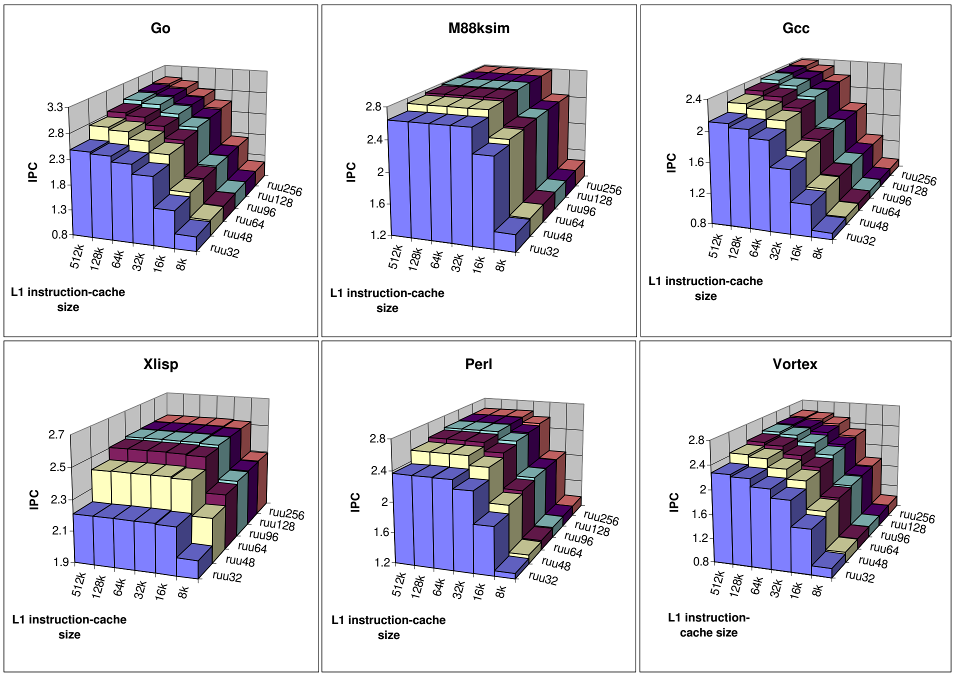


Figure 3.6: IPC as a function of instruction cache size and RUU size with the 100%-direction predictor.

predictors respectively; RUU size in these tables is 64 entries. Note that cache miss ratios include wrong-path accesses, and recall that caches are 2-way set-associative. The higher miss rates in Table 3.3 (100% prediction) result from the lack of pollution by wrong-path prefetching.

| | 8 K | 16 K | 32 K | 64 K | 128 K | 512 K | 2 M |
|---|---|---|---|---|---|---|---|
| go | 0.098 | 0.044 | 0.011 | 0.005 | 0.001 | 0.000 | 0.000 |
| m88ksim | 0.078 | 0.011 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| gcc | 0.126 | 0.069 | 0.029 | 0.008 | 0.003 | 0.000 | 0.000 |
| xlisp | 0.009 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| perl | 0.084 | 0.029 | 0.005 | 0.001 | 0.000 | 0.000 | 0.000 |
| vortex | 0.179 | 0.060 | 0.018 | 0.009 | 0.002 | 0.000 | 0.000 |

Table 3.2: Instruction cache miss ratios, baseline hybrid predictor, 96-entry RUU.

| | 8 K | 16 K | 32 K | 64 K | 128 K | 512 K | 2 M |
|---|---|---|---|---|---|---|---|
| go | 0.143 | 0.068 | 0.017 | 0.007 | 0.002 | 0.000 | 0.000 |
| m88ksim | 0.077 | 0.012 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| gcc | 0.146 | 0.079 | 0.032 | 0.008 | 0.003 | 0.000 | 0.000 |
| xlisp | 0.010 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| perl | 0.095 | 0.032 | 0.006 | 0.000 | 0.000 | 0.000 | 0.000 |
| vortex | 0.162 | 0.056 | 0.019 | 0.009 | 0.002 | 0.000 | 0.000 |

Table 3.3: Instruction cache miss ratios, 100% direction-accuracy predictor, 96-entry RUU.

If enough instructions reside in the RUU, some I-cache misses can be partially or completely hidden with waiting instructions. But I-cache misses are strongly clustered: with an 8 K I-cache, from 45 to 70% of misses happen within 2 cycles of the previous miss, and with a 64 K cache, from 40 to 60%. (90% of *perl's* I-misses happen within 2 cycles, but it experiences only a few thousand misses during the simulated interval.) When misses are so strongly clustered, even a full RUU doesn't help much, so frequent I-cache misses introduce many bubbles. Data misses are less clustered, so this problem is not as prevalent for data references, and an RUU can do a better job of compensating for the cache misses.

The most important implication of these results is that small I-caches limit the value of a larger RUU. Figure 3.2 showed that with 100%-direction branch prediction, RUU size strongly affects performance. Those data assume a 64 K I-cache. But in Figure 3.6, shrinking the I-cache to 8 or 16 K largely wipes out RUU effects for *go, gcc, and perl.* The I-cache misses so often that it can rarely build a sufficient buffer of instructions. When it does, clustering of misses means that even a big and full RUU drains, and the processor still stalls.

This is an especially clear example of shifting tradeoffs: the choice of one design or simulation parameter can control the impact of another. For simulation purposes, this means certain parameters can have a profound impact on how realistic the results are.

I-cache size can influence how much effect RUU size has on IPC, but over the range examined, RUU size does not influence how much effect I-cache size has on IPC.

### 3.3.2 Data Cache

In moving from a smaller RUU to a larger RUU, one might expect that the processor's sensitivity to cache performance would diminish, since there are more opportunities to overlap miss times with useful work. This presumes an RUU with enough correctly-speculated instructions. Turning back to Figure 3.1, with the baseline, hybrid branch predictor, this effect is mildly visible when moving from 32 to 48 RUU entries for *gcc, xlisp*, and more strongly over the entire graph for *vortex.* But "useful" RUU occupancy is generally quite low, so a larger RUU has minimal impact.

| | 8 K | 16 K | 32 K | 64 K | 128 K | 256 K | 512 K | 1 M | 2 M |
|---|---|---|---|---|---|---|---|---|---|
| go | 0.073 | 0.034 | 0.014 | 0.005 | 0.002 | 0.000 | 0.000 | 0.000 | 0.000 |
| m88ksim | 0.007 | 0.003 | 0.001 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| gcc | 0.041 | 0.027 | 0.017 | 0.010 | 0.005 | 0.002 | 0.001 | 0.000 | 0.000 |
| compress | 0.127 | 0.116 | 0.103 | 0.084 | 0.058 | 0.025 | 0.004 | 0.003 | 0.003 |
| xlisp | 0.028 | 0.023 | 0.021 | 0.016 | 0.007 | 0.000 | 0.000 | 0.000 | 0.000 |
| ijpeg | 0.049 | 0.035 | 0.011 | 0.008 | 0.007 | 0.006 | 0.005 | 0.005 | 0.005 |
| perl | 0.034 | 0.012 | 0.006 | 0.001 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| vortex | 0.022 | 0.013 | 0.008 | 0.004 | 0.003 | 0.002 | 0.001 | 0.001 | 0.001 |
| tomcatv | 0.277 | 0.183 | 0.046 | 0.037 | 0.036 | 0.036 | 0.036 | 0.036 | 0.036 |

Table 3.4: Data cache miss ratios, baseline hybrid predictor, 96-entry RUU.

| | 8 K | 16 K | 32 K | 64 K | 128 K | 256 K | 512 K | 1 M | 2 M |
|---|---|---|---|---|---|---|---|---|---|
| go | 0.084 | 0.037 | 0.015 | 0.005 | 0.002 | 0.000 | 0.000 | 0.000 | 0.000 |
| m88ksim | 0.008 | 0.003 | 0.001 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| gcc | 0.045 | 0.029 | 0.019 | 0.010 | 0.006 | 0.002 | 0.001 | 0.000 | 0.000 |
| compress | 0.124 | 0.110 | 0.094 | 0.074 | 0.051 | 0.022 | 0.004 | 0.003 | 0.003 |
| xlisp | 0.030 | 0.025 | 0.022 | 0.018 | 0.007 | 0.000 | 0.000 | 0.000 | 0.000 |
| ijpeg | 0.050 | 0.036 | 0.012 | 0.008 | 0.007 | 0.006 | 0.005 | 0.005 | 0.005 |
| perl | 0.035 | 0.012 | 0.006 | 0.001 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| vortex | 0.022 | 0.013 | 0.008 | 0.004 | 0.003 | 0.002 | 0.001 | 0.001 | 0.001 |
| tomcatv | 0.264 | 0.174 | 0.045 | 0.036 | 0.036 | 0.035 | 0.035 | 0.035 | 0.035 |

Table 3.5: Data cache miss ratios, 100% direction-accuracy predictor, 96-entry RUU.

| | | | |
|---|---|---|---|
| go | 64k | ijpeg | 16k |
| m88ksim | 64k | perl | 64k |
| gcc | 512k | vortex | 128k |
| compress | 512k | tomcatv | 32k |
| xlisp | 512k | | |

Table 3.6: Data cache working sets

The effect is prominent, however, with 100%-direction prediction in Figure 3.2: every benchmark except *ijpeg* exhibits declining D-cache sensitivity as RUU size becomes deeper. For most benchmarks, D-cache hardly matters once the RUU is in 96–256 entries deep, depending on the benchmark.

Plotting performance as a function of cache size, as in Figures 3.1 and 3.2, also permits estimation of the programs' working set sizes. Table 3.6 identifies the point at which further increases in cache size stop producing improvements in IPC—a good approximation of working set size. (Although some applications have a hierarchy of working sets [79].) Knowledge of working set sizes like this can be useful in three ways. It ensures that when choosing a single data cache size for some other type of study (*e.g.* a branch prediction study), the chosen operating point does not have an artificially high miss rate. Second, it helps establish a suitable range of data cache sizes for simulation-based studies [111]. Third, knowledge of working set sizes helps when scaling cache or problem sizes.

Tables 3.4 and 3.5 present data cache miss ratios for the baseline and 100% predictors respectively. Recall that caches are 2-way set-associative. The tables verify that IPC closely correlates with L1 D-cache miss ratio.

### 3.3.3 Instruction Cache and Data Cache Interactions

We have so far examined interactions between data cache size and RUU size, and instruction cache size and RUU size. Figures 3.7 and 3.8 plot IPC as a function of data cache and instruction cache size together. The results follow naturally from the preceding results. Performance is strongly sensitive to I-cache size until the program fits or almost fits into the I-cache. Until the program fits in the I-cache, I-cache size matters substantially more than D-cache size. With 100%-direction prediction, data

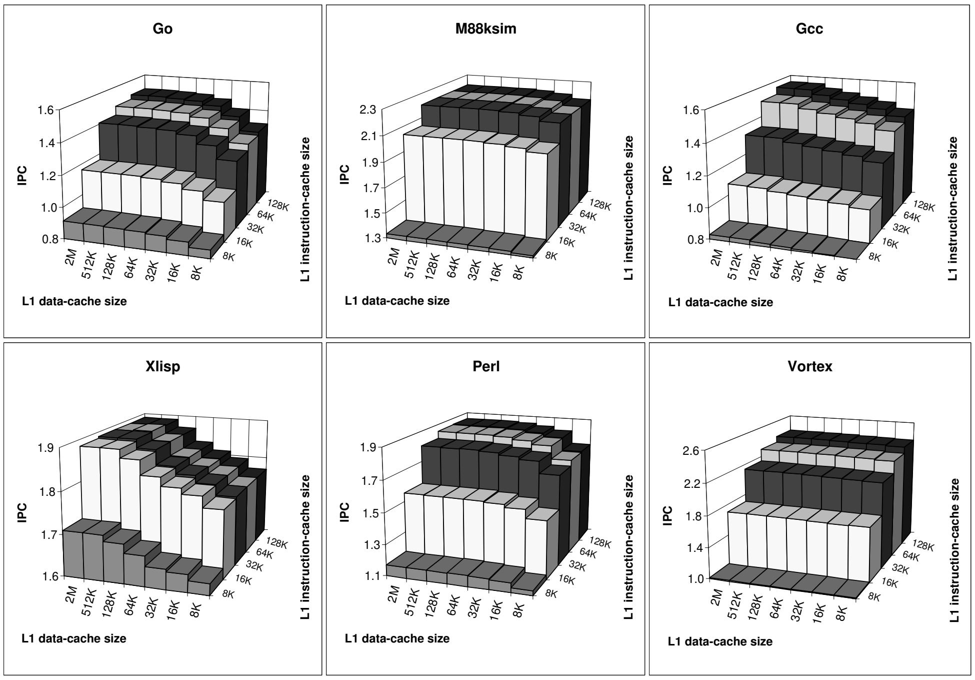


Figure 3.7: IPC as a function of data cache size and instruction cache size with the baseline, hybrid branch predictor. RUU size here is 128 entries.

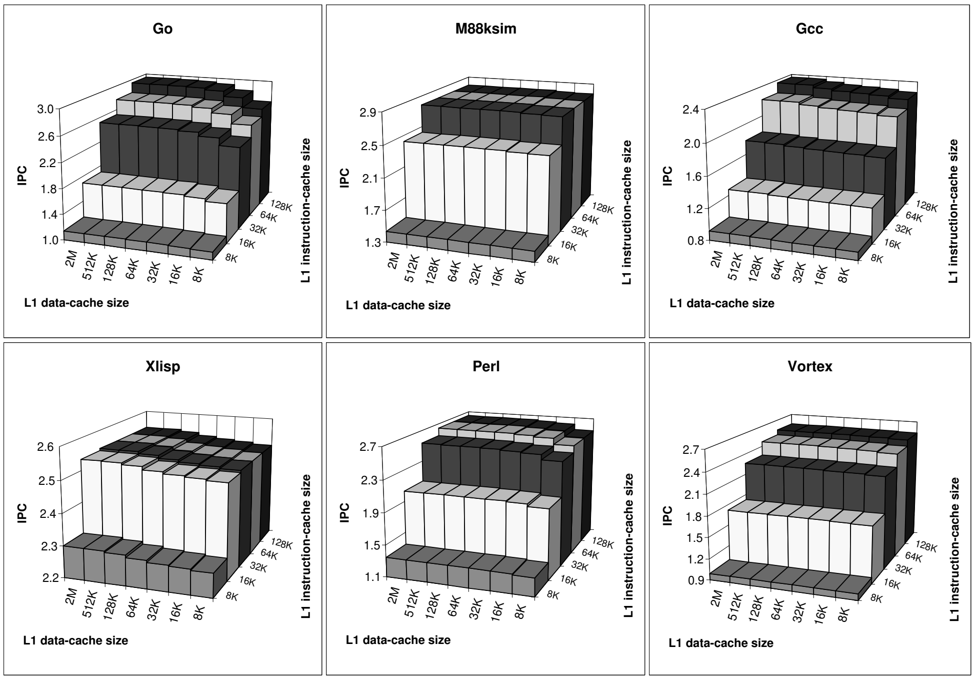


Figure 3.8: IPC as a function of data cache size and instruction cache size with the 100%-direction predictor. RUU size is 128 entries.

cache size matters little because such a large RUU is modeled here. When similar experiments are run with RUU size set to 32 entries, performance is more sensitive to data cache size, just as the "ruu32" data in Figure 3.2 suggests.

Just as too-small I-caches minimize the impact of changing RUU size, small I-caches limit the impact of larger D-caches. This trend is only visible in Figure 3.7; a 128-entry RUU is used here, and the D-cache curves are thus already flat in Figure 3.8. *Go, gcc*, and *perl* especially show how a too-small I-cache limits the benefit of larger D-caches. Many data and instruction misses presumably coincide, and the effect of eliminating some data misses is hidden if the instruction misses remain.

## 3.4 Branch Prediction and the Relative Importance of RUU and L1 Data Cache

Sections 3.2 and 3.3 compared the effects of RUU size and L1 data cache size on performance, and considered both a realistic hardware branch predictor and a 100%-direction-accuracy predictor. The IPC vs. RUU size vs. data cache size graphs in Figures 1 and 2 form the backbone of those observations. The change in moving from the baseline hybrid predictor to 100%-direction accuracy is substantial, and the corresponding IPC surfaces change dramatically between those two figures. This section therefore considers how the surfaces change as branch prediction accuracy increases in finer steps.

Unfortunately, realistic hardware often cannot achieve accuracies near 100%: some branch behavior is simply too random. To see what might happen to RUU-cache tradeoffs in this range of accuracies, this experiment boosts the PHT's accuracy by randomly choosing some mispredictions and *artificially* correcting them. Conditional branch predictions modified in this way behave as though their directions had been correctly predicted. The simulator can be set to attain any desired direction-prediction accuracy by adjusting the fraction of repaired predictions. This artificial adjustment takes place infrequently, so most of the PHT's predictions are untouched. I do not claim here that a program would behave exactly as shown, since the mispredictions to fix up are chosen randomly. The goal is only to suggest how the tradeoffs between RUU size and L1 data cache size might change as branch prediction techniques continue to improve. In fact, the tradeoffs change in a quite clear-cut fashion.

For the first boosting of prediction accuracy, a huge but realistic PHT is used, instead of applying the artificial-boosting technique. This huge predictor is similar to the baseline predictor, but modified to XOR the history bits with some address bits, and with 64 K entries in each table. Subsequent steps in accuracy are achieved with synthetic boosting, but using the huge predictor minimizes the number of correct predictions that need to be artificially created. At 100% direction accuracy, only BTB misses remain.

Figure 3.9 presents graphs of IPC vs. RUU size vs. L1 data cache size for *gcc* over a range of direction-prediction accuracies. The prediction accuracies are indicated on each graph, and all the data are plotted on the same vertical scale. The first graph uses the baseline predictor and is taken from Figure 1; the second-to-last uses 100%-direction prediction and is taken from Figure 2; the intervening graphs use the mechanisms just described. The last graph in Figure 3.9 goes beyond 100%-direction accuracy by correcting all BTB misses, including those for returns and indirect branches—*i.e.*, oracle prediction. Only data for *gcc* are presented, as it is representative of the other benchmarks.

As *gcc*'s direction-prediction accuracy increases from 86% to 100%, not only does overall performance increase, but successive steps in RUU become useful (albeit

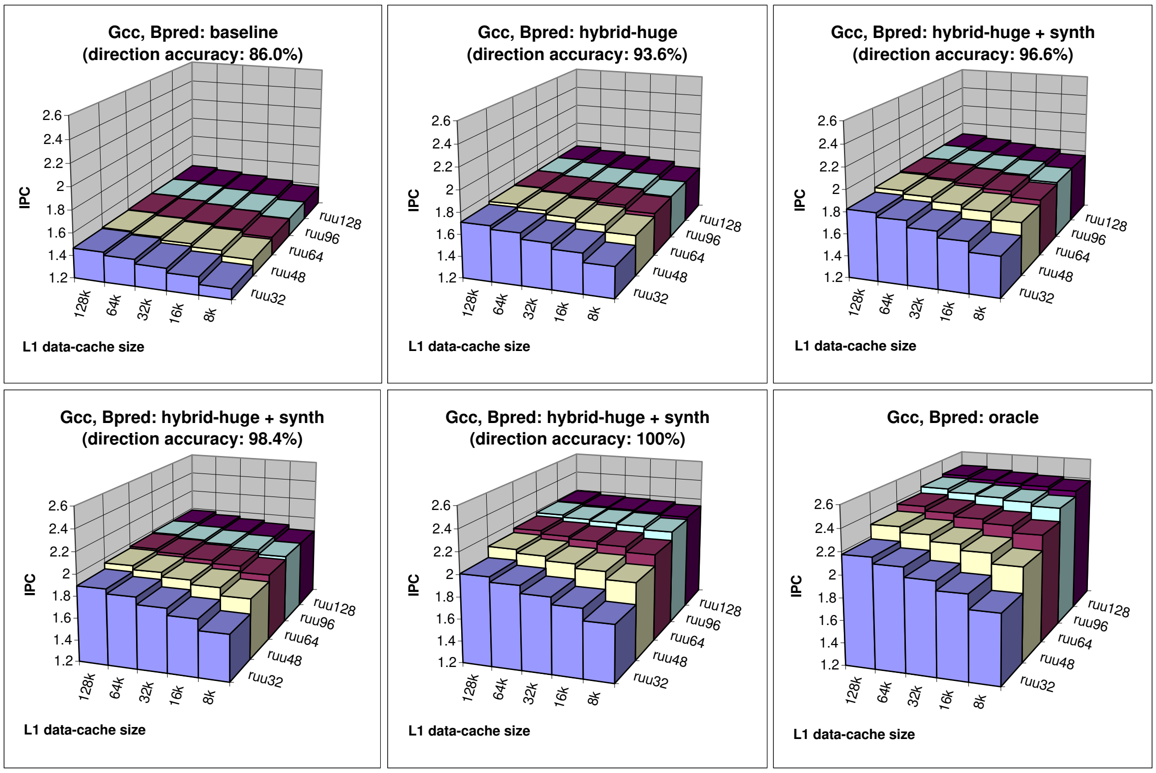


Figure 3.9: Gcc's performance as a function of RUU size and L1 data cache size for a variety of branch prediction accuracies.

slowly), contributing to the increase in IPC. Furthermore, as mentioned earlier, with good branch prediction accuracy, bigger RUUs do such a good job of overlapping cache miss latencies that data cache size becomes less critical. The combination of these effects mean that the slopes along the RUU and cache axes are roughly equal at 98.4% accuracy. In fact, the slope along the cache axes flattens more slowly for *gcc* than for other benchmarks. Conversely, big caches miss so infrequently that big RUUs are not as necessary for hiding misses.

## 3.5 Related Work

Latency tolerance and branch prediction have been studied extensively as separate architectural issues and have each been considered with respect to their interactions with ILP. A sampling of relevant work is given here. Some of the most basic work in the field of superscalar processors has focused on identifying the limits of ILP in different applications. For example, Wall and Jouppi have examined this issue for a variety of realistic and idealized machine configurations [43, 106]. Other work by Smith, Johnson, and Horowitz has also explored the limits on instruction issue afforded by a suite of integer and scalar floating point applications [92]. These works have focused on inherent parallelism in the application. Lam and Wilson have explored the impact of control flow and showed how relaxing control dependence constraints potentially improves performance [55]. Woo *et al.* have characterized the behavior of the SPLASH-2 suite in terms of sensitivity to various parameters and discussed methodological issues for simulating parallel applications [111]. Maynard *et al.* have shown that results using any particular suite of benchmarks must be interpreted with care. For example, SPEC programs have small text sizes, execute little system code, and execute minimal random I/O, while many commercial programs have large text sizes, execute a substantial amount of system code, and perform a substantial amount of random I/O [63]. Many commercial programs also have a large static branch footprint, while most SPEC programs do not [17].

Other fundamental research has focused on understanding and improving branch prediction accuracy via both hardware and software means. Lee, Chen, and Mudge [57], Sechrest, Lee, and Mudge [82], and Chapter 4, among others, have described and measured the different causes of mispredictions, and Evers *et al.* [21] recently measured the benefits of branch correlation. Work by Yeh and Patt [115], Pan *et al.* [70], McFarling [64], Sprangle *et al.* [94], and Eden and Mudge [16], among others, have proposed hardware mechanisms for keeping multi-level branch predictors, for tracking correlations between branches, and for avoiding contention among branches in the predictor's state tables. Such hardware branch prediction mechanisms have been widely incorporated into commercial designs [31, 49, 67]. Some work has also explored software-based branch prediction techniques: Young, Gloy, and Smith [116, 117] have demonstrated compiler-based methods for correlated branch prediction, while Mahlke and Natarajan [58] and August *et al.* [3] have examined branch prediction synthesized in the compiler. Rotenberg *et al.* [39, 78] take an even more aggressive tack: they reorganize the processor around *traces*, groups of basic blocks which have been coalesced into a single unit. When the fetch engine hits in the trace cache, it can provide several basic blocks every cycle without the need for merging cache blocks, multiple-branch or multi-ported predictors, or a multi-ported instruction cache. On a more theoretical level, Gloy and Emer have developed a general language for describing predictors, and they show how it can be used for automated synthesis of predictors. Resulting structures can be complex, but this model may yield insight into avenues for further improvement [19].

While branch prediction is a well-known performance "lever," its relationship to cache design decisions has not previously been quantitatively evaluated. Jouppi and Ranganathan find in [42] that branch prediction is a stronger limitation on performance than memory latency or bandwidth.

Finally, cache design has been a key issue with processor architects for several years now. Many papers study the tradeoffs between L1 cache size and speed; the most recent, simulating a MIPS R10000 model, is by Wilson and Olukotun [110]. Prefetching helps tolerate load latencies, but under out-of-order execution must take into account that many misses are adequately tolerated without prefetching. Most techniques [68, 73] were developed with simpler processor models in mind, but [80], for example, discusses data prefetching for the HP PA-8000. Farkas *et al.* [22] have recently provided some insights regarding memory system design for dynamically-scheduled processors, and Johnson and Hwu [41] discuss a cache allocation mechanism to prevent rarely accessed data from displacing frequently accessed lines. Srinivasan and Lebeck [95] measure loads' latency tolerance and demonstrate the importance of quickly completing loads that feed branch instructions. These papers do not touch on the relationship of branch prediction to cache design as this work does.

## 3.6 Conclusions

By presenting a database of simulation results for the SPECint programs, this chapter examines shifting tradeoffs among instruction window size, first-level data and instruction cache size, and branch prediction accuracy in high-performance processors. The results show that regardless of cache size, having more than 48 RUU (instruction window) entries yields almost no benefit to performance for many benchmarks. Branch prediction is the obstacle. Deeper entries are not used because mispredictions occur often enough to prevent the RUU from building up a large pool of instructions. Even when the deeper entries are active, they usually contain only mis-speculated instructions. This falls far short of the optimum: a great deal of instruction-level parallelism is not being exploited. For programs with branch prediction accuracies near 100%, or if branch prediction could be made perfect, adding RUU even out to 256 entries yields strong benefits.

For many SPECint programs, L1 data cache size is also a strong lever on performance. This becomes less true as branch prediction improves and deeper RUU entries become useful, because the deeper RUU affords enough lookahead to overlap L1 misses with useful computation. At the extreme of 100% prediction accuracy, data cache size hardly matters at all. Conversely, as cache size increases, bigger RUUs help less because fewer misses occur and less lookahead is necessary. Nevertheless, as branch prediction accuracy improves, sensitivity to RUU size increases quickly and RUU effects eventually dwarf L1 data cache effects.

The picture is somewhat different for L1 instruction cache misses, which are typically so tightly clustered—often only 1 or 2 cycles elapse between misses—that even with 100% prediction and a deep window of instructions, a too-small I-cache is the dominant bottleneck.

The most important bottleneck nevertheless remains branch prediction. L1 data-miss penalties will always inspire innovations, but caches are now becoming sufficiently big and sophisticated that future work should perhaps focus specifically on latency-intolerant misses and on better branch prediction. As architects attack the branch prediction bottleneck with more sophisticated hardware schemes and alternative techniques—trace caches [77], compiler-enhanced hardware prediction [3, 58], predication [40, 75], and multipath execution [51, 52, 107] and Chapter 7—larger RUUs will become attractive.

Because latencies can overlap or compound each other in modern, out-of-order processors, design parameters interact in sometimes complex ways. This chapter has illustrated a number of resulting tradeoffs, and just as importantly, has highlighted some potential pitfalls that can result from unwise combinations of branch prediction, instruction window, and cache configurations. The comprehensive data presented here have three main contributions. First, they quantitatively show the importance of considering these configuration issues in conjunction, rather than choosing sizes individually or independently. Second, they help cull the design space to avoid expensive or methodologically flawed simulations. Third, they show the importance of further work on branch prediction. To better understand where further branch-prediction work can be most beneficial, the next chapter develops a *taxonomy* of the different causes of branch mispredictions. The results then motivate the techniques described in the rest of this dissertation.

# Chapter 4

# A Taxonomy of Mispredictions and the Benefits of Alloyed History

## 4.1 Introduction

Predicting the direction of conditional branches has received intense study in recent years, much of it focusing on dynamic, two-level predictors that use history about prior branch outcomes [70, 115]. Yet branch mispredictions remain an important performance bottleneck, because each misprediction results in five, ten, or more cycles of wasted instruction fetch. Even in out-of-order execution systems, mispredictions usually create pipeline bubbles, because the instruction window can rarely expose sufficient ILP to overlap these long penalties. Ranganathan and Jouppi predict that branch prediction will be the most important bottleneck in processors by 2010, worse even than the memory system [42]. Mispredictions also waste power on wrong-path instructions that are eventually squashed [61].

Most proposed predictor organizations today are either refinements of two-level predictors [57, 64, 66, 94], or hybrid organizations combining two or more components [20, 30, 64]. By tracking branch history, two-level predictors can identify branch behavior patterns that a *bimodal*, single-level table—one bias counter per branch site [90]—cannot expose. The history can be either *global*, tracking outcomes from different branches in the same history register, or *local*, tracking history separately for each branch. Global history can expose correlation among branches, while local history is well suited for branches that follow a consistent pattern. In a hybrid organization like the one in Figure 4.1, the components are themselves often two-level components; in this way, the overall predictor can make both global and local history available to the program.

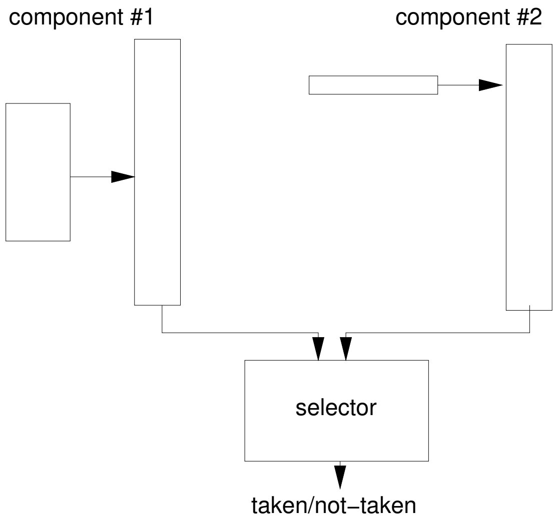


Figure 4.1: The organization of a hybrid predictor with two components. If the hybrid predictor statically assigns branches to one or the other component, the selector simply chooses the appropriate component depending on the branch's PC.

Both organizations have drawbacks. If designers choose a single two-level predictor, they must determine whether to use global or local history—but for most programs, there will be some branches where the choice leads to poor performance. The results presented here show the importance of such *wrong history* mispredictions by presenting a *taxonomy* of misprediction sources. This taxonomy also identifies destructive interference in the table of bias counters (the *pattern history table* or

PHT), long training times, and even mispredictions from intrinsically hard-to-predict branches as other important contributors to programs' misprediction rates. This taxonomy unifies and extends prior efforts to characterize mispredictions [10, 21, 45, 57, 64, 82, 94, 96, 115, 116]

Hybrid organizations are one way to address wrong-history mispredictions. Hybrid predictors make both local and global history available. Because they accomplish this by including multiple components, they also require a selection mechanism to choose which predictor to believe at any moment. Whether static or dynamic, the selector can fail, especially when individual branch sites frequently alternate between using the two types of history. A static selector necessarily encounters mispredictions for these branches, and a dynamic selector, in addition to requiring additional space, can have difficulty learning how to steer branches to the correct component.

**Alloying.** This chapter uses the taxonomy it develops and the resulting data to propose a pseudo-hybrid organization that *alloys* global and local history in the same predictor index, attacking the drawbacks of both organizations. This is a small change to the organization of a two-level predictor, yet it makes both local and global history available and thus provides hybrid-like capabilities without the need for a selection mechanism. Figure 4.2 shows the proposed organization.

The proposed alloyed organization—which I term MAs—is then compared against two well-established, two-level predictors: GAs [70] and PAs [115].[1] Because history patterns often fail to uniquely assign branches to a particular bias counter, GAs and PAs reduce PHT interference by sacrificing some history bits for the anti-aliasing ability of address bits. Alloying can use its richer mixture of bits to accomplish the

[1]In this naming scheme, the first letter indicates the type of history (G = global; P = per-address or local), the second indicates whether the PHT is adaptive or static, and the third indicates whether the PHT consists of one global column, or a set of PHTs selected by the address bits. The "M" in MAs stands for "merged history."

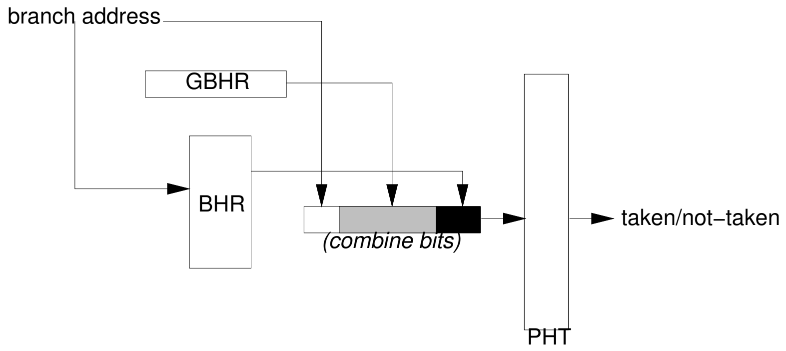


Figure 4.2: The organization of a two-level predictor with an alloyed index. This predictor combines local history from the per-branch history table (BHT) and global history from the global branch-history register (GBHR) with some address bits to compose the PHT index.

same effect, and because it uses less of each type of history, tends to exhibit shorter training time, especially compared to GAs. Alloying's richer mixture of bits also makes it more scalable to large hardware budgets: adding more of any single type of information produces diminishing returns, but MAs uses three types of information.

As mentioned earlier, a number of studies have proposed refinements to two-level prediction—especially to global-history-based predictors. These primarily focus on reducing PHT interference. This study examines the older GAs and PAs organizations and bases MAs upon them. Because all three compose their index via concatenation, the comparison is fair, and all three would benefit similarly from the newer techniques.

This work also compares alloying to hybrid prediction. Alloying eliminates selection failures—there is no longer any selection to be made—but suffers from a corresponding rise in interference within its unified structure.

Generally speaking, alloying represents a combination of two types of previously orthogonal branch prediction research. Proposed enhancements to two-level predictors have used exclusively local or global history. Work describing possible hybrid organizations has provided both types of history, but by using multiple existing two-level predictors and selecting among them. Alloying combines these approaches, as depicted in Figure 4.3. One other study is noteworthy in this regard: the application by Emer and Gloy [19] of genetic algorithms to grow branch predictor structures. This technique can combine different types of information by applying genetics-like mutations to the predictor structure, but seems not to have yielded a structure merging global and local history.

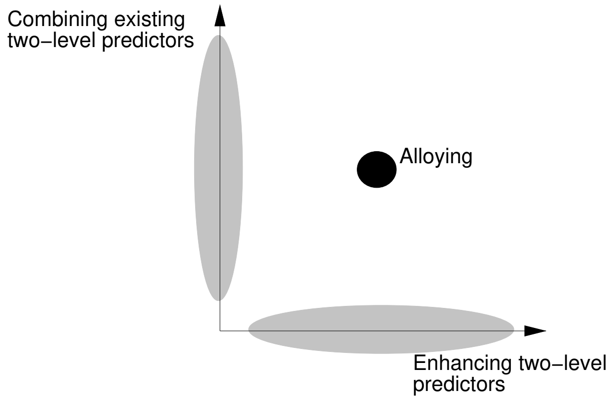


Figure 4.3: Alloying in relation to other proposed history-based branch prediction organizations.

### 4.1.1 Contributions

Specifically, this chapter:

- Develops a taxonomy of misprediction types and extends prior work showing the importance of PHT interference and wrong history.
- Shows that not only do programs as a whole need both global and local history, but so do individual branches.
- Proposes MAs, which alloys both global and local history in the same PHT index. When compared both against solo two-level predictors and against hybrid

predictors, MAs provides excellent performance for a range of predictor sizes. In particular, MAs is the best overall choice for space- and power-constrained environments like embedded processors.

The rest of this chapter is organized as follows. The next section presents the taxonomy of mispredictions, and Section 4.3 then describes the simulation environment and benchmark suite used for the work in this chapter. Section 4.4 uses the taxonomy to compare MAs against GAs and PAs. Then, because a hybrid predictor is another way to make both global and local history available, Section 4.5 compares alloying to two varieties of hybrid prediction and Section 4.6 briefly compares small sizes of all the predictor organizations. Section 4.7 presents related work, and Section 4.8 reviews the chapter's results. Note that this work also appears as [88].

## 4.2 Taxonomy

As mentioned, many studies have measured the importance of PHT interference as a source of mispredictions [10, 57, 64, 66, 82, 94, 96, 116], and other work has shown the importance of wrong-history mispredictions [21]. This section unifies the different misprediction types in a single taxonomy, includes the effects of training time [45], and describes a one-pass method for counting them. This taxonomy also estimates how often branches require both local and global history simultaneously. Note that to make the development of a taxonomy tractable, the categories described here focus specifically on mispredictions in history-based predictors like GAs, PAs, or MAs for a fixed history length.

**Destructive PHT and BHT conflicts.** All three predictors—GAs, PAs, and MAs—can suffer from PHT interference: different branches that map to the same

PHT counter yet have opposite directions. A branch that mispredicts in a baseline organization (GAs, PAs, or MAs) but predicts correctly with an interference-free PHT (GAp, PAp, or MAp) is an instance of *destructive interference*.[2]

Aliasing in the BHT can also cause destructive or constructive interference. The BHT's design incorporates some additional variables that are beyond the scope of this taxonomy, so here an interference-free BHT is assumed. Section 4.4.2 examines overall prediction accuracies when using a finite, set-associative BHT.

**Training-induced mispredictions.** If a misprediction is not caused by PHT interference, it can instead occur because the predictor has not yet learned the branch's behavior. This happens especially at the beginning of a program or after a context switch, but also occurs as programs transition from one phase to another. I have yet to devise a precise method for measuring training mispredictions, but propose that their contribution can be estimated with a bimodal predictor as follows.

PHT conflicts must first be eliminated as above. Then, if a branch mispredicts in a particular organization but predicts correctly in a simpler organization, the former has not yet fully learned the branch's behavior. The simpler predictor used here is a bimodal predictor.

If a branch mispredicts in an $n$-bit-history GAp, PAp, or MAp predictor but predicts correctly in an $n-1$ bit GAp or PAp predictor, the misprediction occurred either because the $n$ bit predictor has not yet reached steady state for the branch in question, or possibly due to a lucky guess by the $n-1$ bit predictor. Training mispredictions in an $n-1$ bit predictor can in turn be identified with an $n-2$ bit predictor. The extreme, an $n-n$ bit predictor, is the bimodal predictor. To avoid this potentially prohibitive sequence of tests, I collapse them: any misprediction that

[2] Note that constructive interference can also occur, so the expected gain from eliminating PHT conflicts would only be the difference between the two values.

does not result from a conflict and which the bimodal predictor can eliminate, I deem a *training misprediction*. This is only an approximation.

**Wrong type of history.** Mispredictions can also occur because the predictor tracks the wrong type of history for the branch in question: global instead of local, or vice-versa. The measurements here separate "true" wrong-history mispredictions from those merely caused by interference. True wrong-history misses occur when a non-conflict-miss/non-training-miss is correctly predicted using the other type of history. These mispredictions can potentially be completely eliminated by MAs, because it provides both types of history.

**Needs both types of history.** For some branches, neither type of history alone suffices; instead the branch *needs both* types of information simultaneously. The frequency of such mispredictions can be estimated using an MAs predictor, inferring that mispredictions which do not fall into the preceding categories and which MAs can eliminate are need-both mispredictions. The number of need-both mispredictions is always the same for both GAs and PAs.

**Taxonomy measurement.** Figure 4.4 shows the sequence of tests just described. Each dynamic branch can be categorized for both GAs and PAs by following the sequence of tests described until it predicts correctly, or falls through as a misprediction that could not be categorized. These *remaining* branches are either inherently difficult to predict, or need longer history.

An advantage of this taxonomy is that this cascaded sequence of tests results in a set of mutually exclusive categories. Each misprediction falls into exactly one category. Some mispredictions might fall into different categories were the sequence of tests altered. If history type (local vs. global) were now tested first, that might

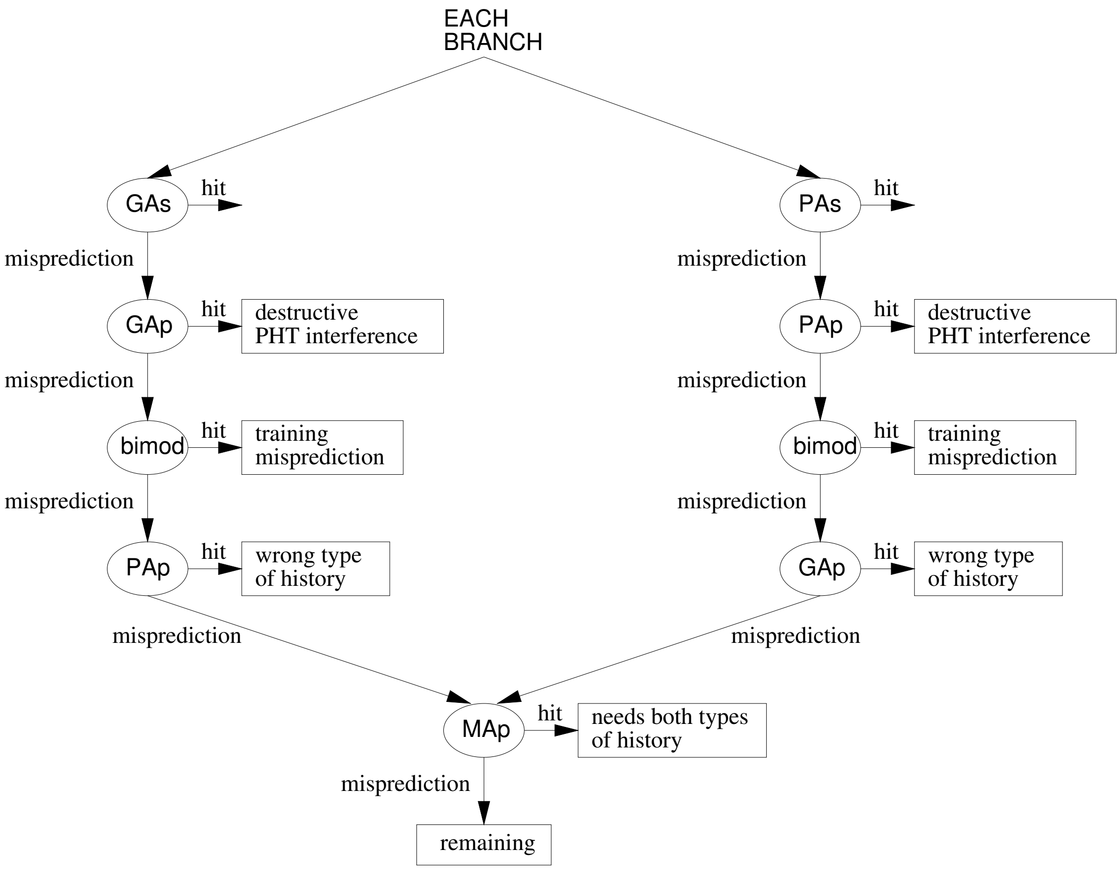


Figure 4.4: A flowchart depicting how the taxonomy categorizes misprediction types. Each dynamic branch flows down both sides until it is either labeled or falls through.

capture some mispredictions that were categorized as conflicts using the original sequence. But this happens because changing history types changes the aliasing behavior of the branches. The sequence described in this section is therefore the proper one. Changing the PHT or BHT associativity does not influence a branch's need for history type. Indeed, testing for destructive conflicts first captures mispredictions that would otherwise *appear* to be wrong-history mispredictions, but only because using the alternative type of history changes aliasing behavior and removes a destructive conflict. Similar arguments lead to the order presented here.

Finally, note that conflict and training mispredictions can also be categorized for

MAs using an MAp predictor and a bimodal predictor.

**Improper update timing.** A final category that is not evaluated here but is a vital component of the misprediction rate results from improper update timing. Late update after a branch resolves—non-speculative update—makes the branch predictor's state stale and possibly inaccurate, which results in mispredictions. Speculative update risks placing incorrect state into the predictor if a misprediction occurs, but simple *fixup* mechanisms can correct any corruption that might result and improve performance by 10% for a global history predictor and 5% for a local history predictor (see Chapter 5). Similar issues arise for the return-address stack (see Chapter 6). Update-timing effects are not evaluated here because they require detailed, cycle-accurate simulations, while in the interests of simulation speed and simulation length, this framework uses instruction-level simulation as described in the next section.

## 4.3 Simulation and Benchmark Details

The simulation methodology used for this study differs from that used in the other chapters, so that methodology is described next. Then the next section compares GAs, PAs, and MAs using the previously described taxonomy.

### 4.3.1 Simulator

These simulations use the instruction-level branch predictor simulator from the University of Wisconsin's SimpleScalar toolkit v2.0 [5], instead of the detailed cycle-level simulator. This permits much longer simulation times. Simulations still capture all non-kernel behavior, including library code.

The results in this chapter only report conditional-branch accuracies. Unconditional branches do not use the direction-prediction portion of the branch predictor. Their performance depends on the branch target buffer's (BTB) behavior, whose design is beyond the scope of this work. Prediction accuracies are therefore reported in terms of the percentage of conditional branches whose *direction* has been correctly predicted, regardless of whether they miss in the BTB.

Changes in conditional-branch prediction accuracy do not necessarily translate directly into corresponding changes in overall performance. That depends on the frequency of branches within the code, the degree of instruction-level parallelism, the average misprediction penalty, and other factors. Nevertheless, all other things being equal, better prediction improves performance. Measurements can therefore be restricted to prediction type and accuracy, in order to focus on different predictors' ability to exploit predictability.

### 4.3.2 Benchmarks

These evaluations use not only the SPECint95 benchmarks [98] described in Chapter 2, but also 4 other primarily integer benchmarks and three SPECfp95 benchmarks. The SPECfp benchmarks have been added to show the impact of these branch prediction techniques on loop-oriented, numeric codes. The 4 other integer benchmarks have been added to show that the results in this work are not specific to characteristics of SPECint programs.

*Gnuchess* comes from the IBS benchmark suite [103] and is GNU's implementation chess. The benchmark sets the computer to play itself and deactivates the X interface. *Wolf* is the timberwolf circuit router and comes from Smith's Unix-Utils benchmark suite [91]. 1.7% of its instructions are floating-point operations. I was unable to simulate other members of the IBS or Unix-Utils suite due to difficulties compiling them for the SimpleScalar PISA or because they use the X library, which is not part of the SimpleScalar 2.0 toolkit.

*Radiosity* and *volrend* were chosen from the SPLASH2 suite [111] of parallel applications for shared memory because these two programs have significant misprediction rates and interesting behavior. These are compiled for a single processor using the null macros. *Radiosity* computes the equilibrium distribution of light in a scene. and *volrend* renders a three-dimensional volume using a ray casting technique.

Finally, three SPECfp benchmarks, *tomcatv, mgrid*, and *hydro2d*, are included. They all have tiny misprediction rates compared to the other applications but are representative of many scientific calculations.

| | Warmup insts | Simulated insts | Conditional branch static sites | Conditional branch dynamic refs |
|---|---|---|---|---|
| go | 925 M | 1 B | 5,331 | 112 M |
| m88ksim | 0 | 1 B | 968 | 162 M |
| gcc (cc1) | 0 | 1.242 B | 20,783 | 190 M |
| compress | 1648 M | 965 M | 203 | 151 M |
| li (xlisp) | 0 | 1 B | 676 | 154 M |
| ijpeg | 823 M | 1 B | 1,415 | 58 M |
| perl | 600 M | 1 B | 614 | 129 M |
| vortex | 2450 M | 1 B | 3,203 | 124 M |
| tomcatv | 2950 M | 1 B | 186 | 130 M |
| mgrid | 475 M | 1 B | 923 | 154 M |
| hydro2d | 375 M | 1 B | 1,205 | 189 M |
| gnuchess | 0 | 1 B | 1,127 | 96 M |
| wolf | 0 | 164 M | 2,993 | 26 M |
| radiosity | 300 M | 1 B | 183 | 92 M |
| volrend | 125 M | 1 B | 660 | 70 M |

Table 4.1: Benchmark summary.
"Warmup insts" indicates the length of the preliminary phase of simulation, before statistics-gathering is activated. Also note that, for compress, only the compress phase is simulated.

Chapter 2 showed that some benchmarks, like *compress*, have substantial initial phases in which they generate data (as in *compress*), read in data, or perform other actions that differ from the main body of the execution. Simulations produce substantially unrepresentative results if this initial phase comprises too much of the simulation. For this reason, the simulator begins gathering statistics only after the initial phase has passed, just as in described in previous chapters. During this preliminary phase of simulation, branches are still presented to the predictor so that it can warm up. A difference in these simulations is that, because programs are now simulating many more instructions (1 billion), benchmarks like *gcc* that have a minimal initial phase omit the simulator warmup phase.

All benchmarks are compiled using `gcc -O3 -funroll-loops` (`-O3` includes inlining). The floating-point programs, which are written in Fortran, are first converted to C using AT&T Bell Labs' *f2c* (1994 version). Most benchmarks are simulated for 1 billion instructions. The SPEC programs use "ref" inputs. Some benchmarks come with multiple reference inputs, in which case one has generally been chosen. For the SPECint benchmarks, these are the same as described in Chapter 2. The exceptions are *xlisp*, which is run with the 9-queens problem (the SPEC92 ref input), and *compress*, which is simulated for an entire execution of its compress phase.

*Gnuchess* is run at a level of 10, and *wolf* is run with the supplied `cbd` input, which can be run to completion. *Radiosity* uses the default parameters and the default "room" model, and *volrend* uses the largest provided input. The benchmarks are summarized in Table 4.1, including the simulator warmup interval, each program's static footprint (the number of branch sites exercised during execution), and the number of dynamic branches executed.

## 4.4 Using the Taxonomy to Compare Two-Level Predictors

In addition to comparing the overall misprediction rates of GAs, PAs, and alloyed (MAs) predictors, this section explores the behavioral differences among the different two-level organizations by breaking down mispredictions into categories using the taxonomy.

### 4.4.1 Performance with Equal PHT sizes

Comparing equal-sized PHTs and hence equal index sizes reveals the "indexing strength" of different possible bit combinations. To get the best comparison, best possible GAs, PAs, and MAs predictors for any particular size must be compared. The brute-force method, simulating all possible combinations of global, local, and address bits for the desired index size, not only identifies the best configurations, but yields surfaces interesting in their own right. These surfaces have been mapped out for *go, m88ksim, gcc*, and *compress*. Figures 4.5 and 4.6 present examples of these surfaces for *gcc* and *m88ksim* with a 32K-entry PHT.

A noteworthy feature of the surfaces is that using hill-climbing to find the best combination can fail due to local optima. One benefit of the taxonomy is therefore that understanding the source of mispredictions can make maneuvering through this non-monotonic space somewhat easier. The problem is pronounced for *m88ksim* when choosing a starting point with a small number of local history bits. Starting with 15 global history bits and asking whether to replace some of them with other types of bits, hill-climbing leads to an index composed of 11 global history bits, 2 local-history bits, and 2 address bits. The true best configurations, however, are 6 bits global and

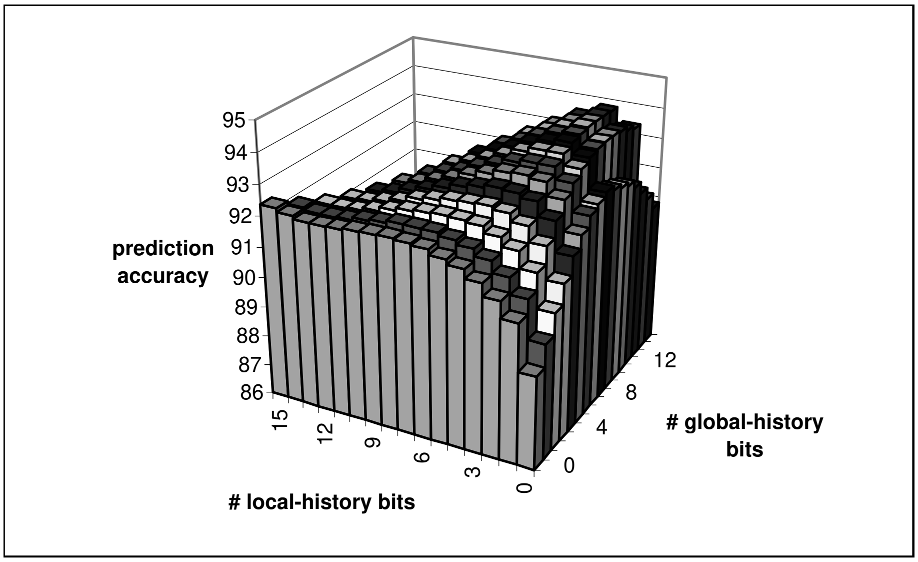


Figure 4.5: A surface showing, for gcc, the prediction accuracy of all the possible combinations of global-history, local-history, and address bits in a 15-bit index. The number of address bits can be found be subtracting the sum of global- and local-history bits from 15. Only indexes totaling 15 bits are simulated, so the back half of this surface (approaching the coordinates (15,12)) is absent, which is not evident from this perspective.

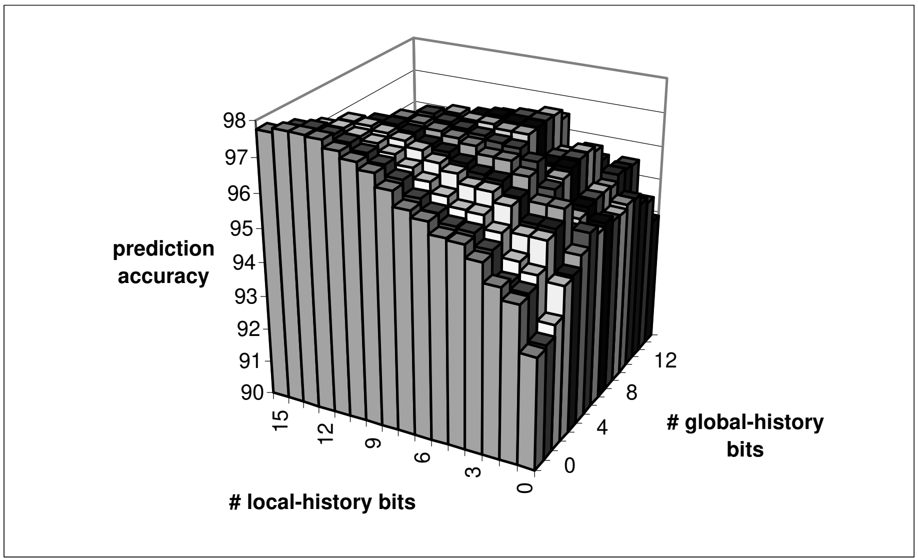


Figure 4.6: A surface showing, for m88ksim, the prediction accuracy of all the possible combinations of global-history, local-history, and address bits in a 15-bit index.

9 bits local, and—a tie—no global, 14 bits local, and 1 bit of address. A similar effect occurs with *gcc* if hill-climbing uses an all-local-bit index as a starting point, but the effect is barely perceptible at this scale.

With these maps of the parameter space, the one configuration that gives the best performance over these programs can be chosen; these choices appear in Table 4.2. In each case, the PAs and MAs predictors use an interference-free BHT so that equal-length indexes can be compared without artifacts from BHT behavior. Figures 4.7–4.9 show both the overall misprediction rate—the bars' total height—and the breakdown of mispredictions for three PHT sizes: predictors with 32 K counters, 4 K counters, and 1 K counters.

| | GAs | PAs | MAs |
|---|---|---|---|
| 32K entries | 8g, 7a | 14p, 1a | 10g, 4p, 1a |
| 4K entries | 5g, 7a | 10p, 2a | 7g, 4p, 1a |
| 1K entries | 1g, 9a | 10p, 0a | 5g, 4p, 1a |

Table 4.2: Predictor configurations used for equal-PHT comparison. "g" indicates the number of global-history bits, "p" local-history bits, and "a" address bits.

The results are strong. For every benchmark and each of the three PHT sizes, the alloying approach—MAs—outperforms GAs. In fact, MAs is usually superior by a substantial margin. MAs also outperforms PAs for most benchmarks. The exceptions are *m88ksim, xlisp, gnuchess*, and *tomcatv*. These benchmarks require substantial self-history, longer than that provided by MAs. This behavior can be inferred from the substantial wrong-history component of the GAs bar (the stippled segment): 47% of *xlisp*'s and 16.5% of tomcatv's GAs mispredictions with a 4k PHT, for example.

The number of mispredictions that "need both" (the white segment of each bar) is small: 5–15% of mispredictions. Compared to GAs, the majority of benefit for MAs

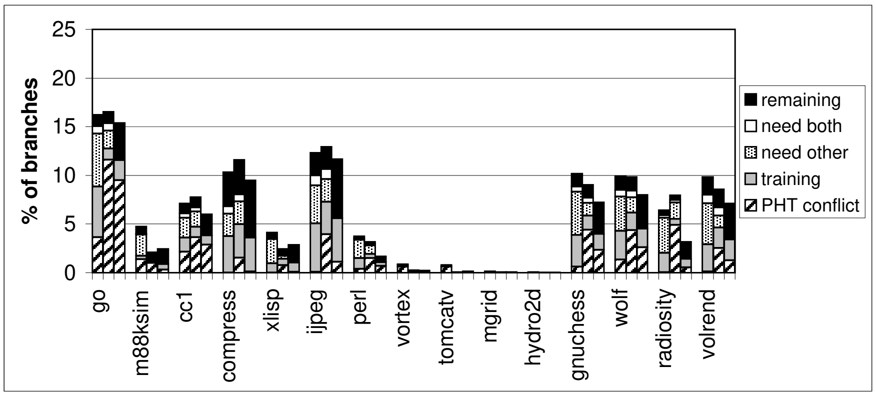


Figure 4.7: Relative performance of GAs, PAs, and MAs for a 32K-entry PHT and an interference-free BHT, including a breakdown of misprediction types. KEY: For each benchmark, the leftmost bar represents GAs, the center bar PAs, and the rightmost bar MAs. Note that "cc1" is the same as gcc.

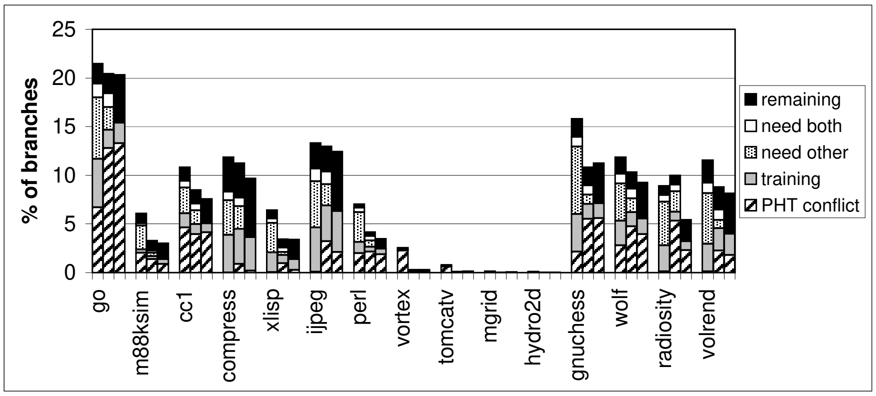


Figure 4.8: Relative performance of GAs, PAs, and MAs for a 4K-entry PHT and an interference-free BHT, including a breakdown of misprediction types.

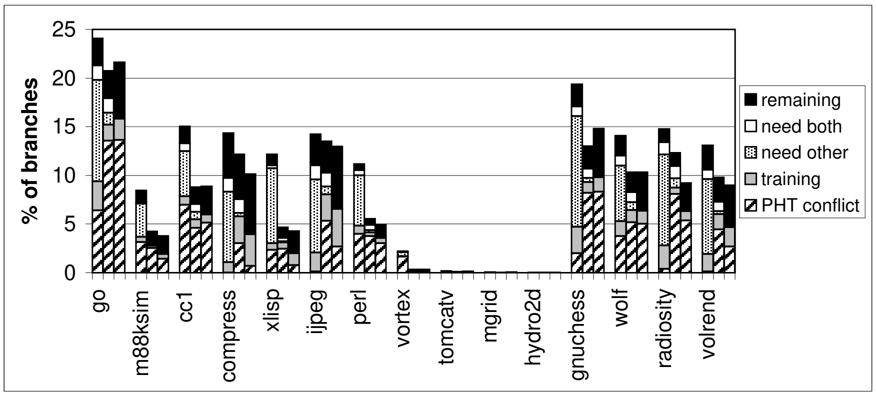


Figure 4.9: Relative performance of GAs, PAs, and MAs for a 1K-entry PHT and an interference-free BHT, including a breakdown of misprediction types.

| | 64 Kbits | | 8 Kbits | | 2 Kbits | |
|---|---|---|---|---|---|---|
| | GAs | PAs | GAs | PAs | GAs | PAs |
| go | 5.14 | 6.99 | 5.26 | 0.54 | 10.23 | -4.34 |
| m88ksim | 48.29 | -15.94 | 50.34 | 8.18 | 55.03 | 10.91 |
| gcc | 15.82 | 22.59 | 30.13 | 10.78 | 40.91 | -1.09 |
| compress | 8.31 | 18.26 | 18.45 | 13.92 | 29.42 | 16.72 |
| xlisp | 30.45 | -18.63 | 46.73 | 0.49 | 64.76 | 8.38 |
| ijpeg | 5.09 | 9.79 | 6.49 | 4.25 | 8.98 | 3.98 |
| perl | 54.71 | 46.30 | 50.68 | 16.33 | 56.00 | 11.58 |
| vortex | 75.23 | 16.67 | 88.69 | 4.26 | 84.21 | 0.00 |
| tomcatv | 80.07 | -133.33 | 79.58 | -44.74 | 16.67 | -35.25 |
| mgrid | 56.93 | 1.67 | 57.25 | -5.36 | 16.90 | -3.51 |
| hydro2d | 65.63 | 12.00 | 76.77 | 4.17 | 27.27 | 7.69 |
| gnuchess | 28.81 | 19.82 | 28.88 | -3.63 | 23.56 | -13.80 |
| wolf | 19.10 | 18.62 | 21.97 | 10.66 | 26.79 | 0.29 |
| radiosity | 50.47 | 60.06 | 38.86 | 45.48 | 37.71 | 25.33 |
| volrend | 27.56 | 16.66 | 29.67 | 7.58 | 31.39 | 8.25 |

Table 4.3: Percentage by which MAs's misprediction rate is better than GAs or PAs for same-sized PHTs.

evidently stems from its ability to address wrong-history mispredictions, which are substantial for every benchmark under GAs. This suggests that many branches need at least a few bits of self-history. Global history does in principle include a branch's own history, but the global history's contents turn over very rapidly. For example, with an 8-bit global history (as used by GAs with a 32k-PHT), the global history turns over so quickly that almost no branches can find any self history in the GBHR.

PAs tracks self-history, but cannot observe correlations among different branch sites, so most benchmarks also exhibit a substantial wrong-history component when using a PAs predictor. Exceptions are *m88ksim, xlisp*, and the three SPECfp benchmarks. For 1 K counters, the global history predictor only uses 1 bit of global history, so in this case few benchmarks indicate any substantial need for global history. When benchmarks have minimal need for global history, PAs performs well compared to

GAs.

Another factor benefiting MAs when compared to PAs is that MAs almost always exhibits less PHT interference. (The only exception is *gcc* for smaller PHTs.) This is partly because my parameter search produced a PAs configuration using a minimal number of address bits: the benefit of additional local history bits outweighs the increased interference. On the other hand, PAs and MAs both usually suffer from more interference than GAs, because the best GAs configuration incorporates so many address bits.

Overall, MAs consistently performs well compared to conventional two-level predictors. It makes both types of history available all the time, and also reduces PHT interference compared to PAs. The degree to which MAs outperforms (or occasionally underperforms) GAs and PAs appears in Table 4.3. For example, with a 32K-entry PHT, MAs cuts the GAs misprediction rate by at least 25% in 10 of the benchmarks, and in 11 of the benchmarks with a 1K-entry PHT. Even for benchmarks which already have a high prediction accuracy, gains of this magnitude are significant.

### 4.4.2 Performance with Equal Overall Area

The preceding section provided strong results for MAs, but used an interference-free BHT and equal PHT sizes. This gave an answer to the question: what is the most productive way to compose a PHT index of some particular size?

MAs and PAs have a substantial advantage with an interference-free PHT, because they have extra information: $n$ bits of local history for every branch. Figures 4.10–4.12 therefore repeat the above experiments, but for equal-size overall configurations.

The configurations used appear in Table 4.4. These were found by halving the PHT size for PAs and MAs in order to make space available for the BHT, and then exploring a number of BHT configurations, ranging from wide and short (many local-history bits and few BHT entries) to narrow and tall. The importance of BHT contention makes BHT height more important than BHT width for both PAs and MAs, so programs generally prefer a tall BHT even though this means a very narrow local-history width. This is true even when using a 4-way associative BHT. (The BHT can use the BTB's tags if they have the same organization.) These configurations are a sharp contrast to those used above, where programs benefited from quite long local histories.

| | GAs | PAs | | | MAs | | |
|---|---|---|---|---|---|---|---|
| | | index | BHT | PHT | index | BHT | PHT |
| 64 Kbits | 8g, 7a | 8p, 6a | 4K entries | 16K entries | 9g, 4p, 3a | 8K entries | 16K entries |
| 8 Kbits | 5g, 7a | 4p, 7a | 1K entries | 2K entries | 7g, 2p, 2a | 2K entries | 2K entries |
| 2 Kbits | 1g, 9a | 2p, 7a | 512 entries | 512 entries | 3g, 2p, 4a | 512 entries | 512 entries |

Table 4.4: Predictor configurations used for equal-total-size comparison. BHTs are 4-way associative and share tags with the BTB.

The impact of contention in a finite BHT can be seen not only in the different configurations used, but in the substantial loss in prediction accuracy for MAs and PAs compared to Figures 4.7–4.9. (The GAs results are of course unchanged.) Although MAs loses some advantage compared to GAs, it remains a winner for most benchmarks, and especially at 64 Kbits. Table 4.5 summarizes the improvement in misprediction rates with MAs over GAs and PAs.

These results show how much further work on reducing BHT contention, like [8, 50], can help push performance toward those in Section 4.4.1.

### 4.4.3 Dynamically Configurable Indexes

The simulations to compare GAs, PAs, and MAs for the above results used the one best overall configuration. Individual benchmarks, however, have varying pref-

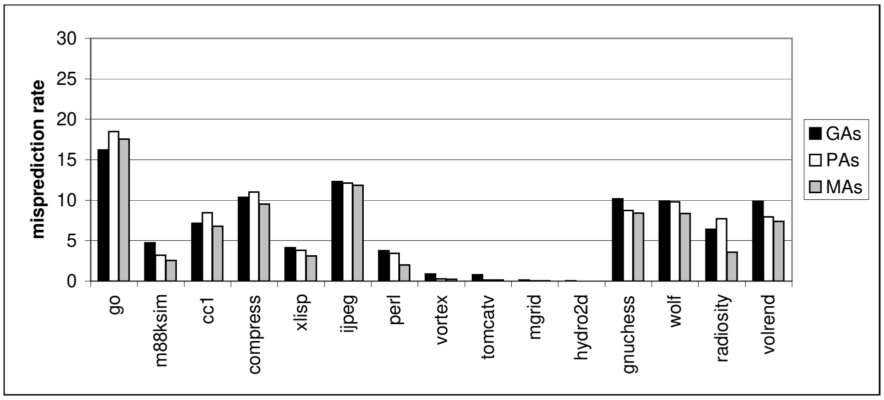


Figure 4.10: Relative performance of GAs, PAs, and MAs for a 64 Kbits total size, including a finite BHT.

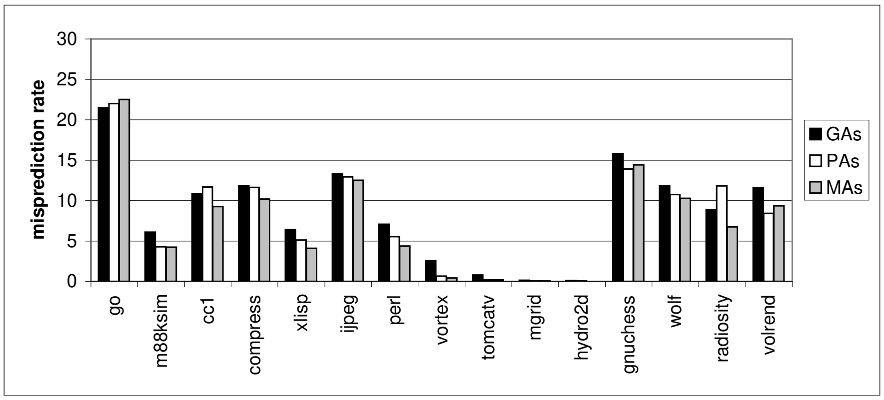


Figure 4.11: Relative performance of GAs, PAs, and MAs for 8 Kbits total size, including a finite BHT.

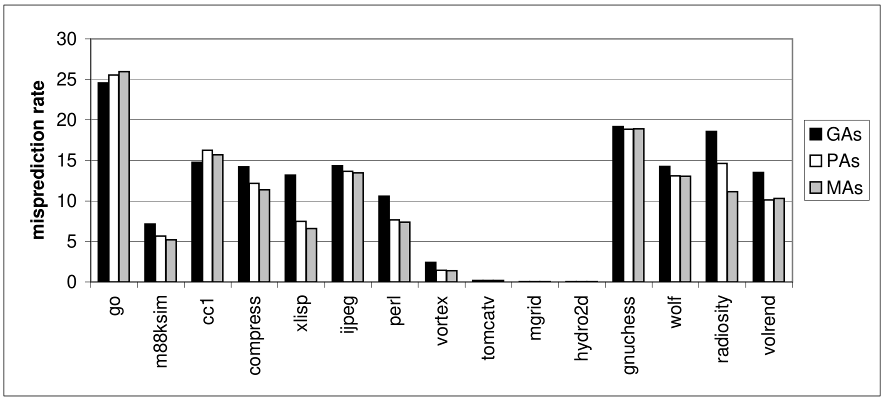


Figure 4.12: Relative performance of GAs, PAs, and MAs for 2 Kbits total size, including a finite BHT.

| | 64 Kbits | | 8 Kbits | | 2 Kbits | |
|---|---|---|---|---|---|---|
| | GAs | PAs | GAs | PAs | GAs | PAs |
| go | -8.02 | 5.09 | -4.78 | -2.30 | -5.60 | -1.49 |
| m88ksim | 46.04 | 19.38 | 30.60 | 1.26 | 27.19 | 8.35 |
| gcc | 5.04 | 19.96 | 14.89 | 20.76 | -6.42 | 3.53 |
| compress | 7.91 | 13.31 | 14.15 | 12.32 | 19.75 | 6.30 |
| xlisp | 24.88 | 18.28 | 36.03 | 19.59 | 49.95 | 11.52 |
| ijpeg | 3.86 | 2.53 | 5.94 | 3.16 | 6.31 | 1.55 |
| perl | 46.96 | 41.89 | 37.81 | 20.66 | 30.59 | 4.07 |
| vortex | 72.12 | 8.33 | 84.42 | 40.36 | 42.76 | 5.70 |
| tomcatv | 80.07 | -11.03 | 78.47 | -4.82 | 11.62 | 0.57 |
| mgrid | 56.93 | -5.36 | 56.52 | -1.69 | -3.28 | 0.00 |
| hydro2d | 64.06 | 0.00 | 76.77 | 4.17 | 20.00 | 0.00 |
| gnuchess | 17.58 | 4.16 | 8.70 | -3.82 | 1.44 | -0.42 |
| wolf | 15.65 | 14.65 | 13.31 | 4.53 | 8.31 | 0.11 |
| radiosity | 44.28 | 53.53 | 24.27 | 42.83 | 39.99 | 23.84 |
| volrend | 24.99 | 6.93 | 19.32 | -10.94 | 23.88 | -1.85 |

Table 4.5: Percentage by which MAs's misprediction rate is better (or sometimes worse) than GAs or PAs for equal-size configurations.

erences; indeed, sometimes contradictory preferences. Even though MAs already performs well, this suggests that an ability to dynamically configure the PHT index's composition—a small change—would be helpful. Then on start-up, each program can set the index to the most beneficial bits from the available global history, local history, and address bits.

As an example of the impact, consider the two benchmarks, *gcc* and *m88ksim*, for which Figures 4.5 and 4.6 presented MAs-performance surfaces. They prefer dramatically different MAs configurations. *M88ksim* does best with an almost entirely local-history index, while *gcc* does best with a primarily global-history index. In each case, the best configuration for one gives poor performance for the other. Table 4.6 compares these two cases.

These points are, of course, extreme. Yet most benchmarks prefer an MAs config-

| | misprediction rates | | | Δ rates | |
|---|---|---|---|---|---|
| | m88 best<br>14p, 1a | gcc best<br>2p, 10g, 3a | compromise<br>4p, 10g, 1a | Δ(best–worst) | Δ(best–compromise) |
| m88ksim | 2.12 | 3.32 | 2.46 | 56.60% | 16.04% |
| gcc | 7.75 | 5.7 | 6 | 35.96% | 5.26% |

Table 4.6: Alternative configurations for gcc and m88ksim and their performance. "Compromise" represents the best overall organization found.

uration in the general vicinity of the one shown for *gcc*, and the overall, "compromise" configuration used in Section 4.4.1 is similar to *gcc*'s best. Both benchmarks do fairly well with that compromise (also shown in Table 4.6). But a slight change in the benchmark suite could easily shift the compromise configuration to actually be the best one for *gcc*, which is so poor for *m88ksim*. Some configurations in the immediate vicinity of *gcc*'s best actually cause even worse performance for *m88ksim*: for example, using 13 bits of global history makes *m88ksim*'s misprediction rate nearly double again.

Such a scenario could occur, for example, if *m88ksim* were not in the benchmark suite. This of course argues for care in selecting benchmarks, but no suite can be all-inclusive. This effect is also not restricted to alloyed indexes. Benchmarks exhibit contradictory preferences for PAs and especially for GAs. In the latter case, programs which suffer from little PHT interference prefer mostly global history in the index, while others prefer many fewer global bits and want address bits instead.

Allowing each program to determine the index's composition could therefore yield large benefits for programs that behave differently from the benchmark suite used in evaluating the processor design. Choosing a best configuration entails profiling the program's prediction accuracy on the possible alternatives and selecting the best, a process that can easily be automated. The profiling can be done in parallel for all possible configurations, using instruction-level simulations (like those used for this chapter) for each possible mixture of bits. A similar technique could be used to configure the branch predictor differently for individual segments of a particular program. In either case, the profiling need only be done once for each new program. Programs that have not yet been profiled cannot take advantage of the branch predictor's configurability, but profiling a new program is easily done once the process has been automated.

A configurable GAs or PAs organization would most likely provide full history, but allow the program to configure the PHT index to mask out some history bits and replace them with address bits. Note that the history-length fitting approach described in [45] can accomplish this dynamically for GAs or PAs—no profiling required—but requires additional hardware to monitor the program's hardware needs. A further concern with this approach is that each change in history length rearranges branches' mapping into the PHT, potentially requiring a new training time. Runtime-adaptive mechanisms like these, however, are of great interest because they eliminate the profiling step, they are not susceptible to variations in program input, and they permit the branch predictor's configuration to vary more finely over time than a static method does.

A configurable MAs organization would perhaps include a BHT of moderate width (4–8 bits; most of the benefit of local history is achieved with 4 local history bits), a full length GBHR, and the ability to choose which bits to include. Some programs might even choose to make MAs behave strictly like GAs or PAs.

More specifically, a configurable predictor would add a two-input or three-input multiplexor to each index bit of the PHT. The possible inputs consist of global history bits, local history bits, and address bits. Three-input multiplexors are only needed for those index bits which might any of the three types of information. The multiplexors would be controlled by a special register, set by the program at startup and included as part of its context. This reduces timing concerns for the selector value, since it will be static between context switches. The only hardware cost is then the added multiplexors and the small associated gate delay.

Alternatively, the multiplexor could be replaced by any combinational circuit with a sufficiently short delay, allowing a more sophisticated operation—like XOR, or perhaps even hardware supporting user-programmable functions—on the input bits.

## 4.5 Alloyed History vs. Hybrid Prediction in High-Performance Processors

The breakdown of misprediction types in Section 4.4.1, as well as MAs's success compared to GAs and PAs, suggest that many programs need a predictor that includes both local and global history. Since only a small fraction of branches benefit from having both types available simultaneously, hybrid predictors that combine both a global history and a local history component can eliminate many of the same mispredictions as MAs, and can possibly do a better job than MAs of attacking the PHT-conflict problem. With a static selector like the one proposed by Grunwald, Lindsay, and Zorn [30], branches only update their assigned component, with a substantial corresponding decrease in interference. With a dynamic selector like those proposed by McFarling [64] and Chang, Hao, and Patt [11], if one component suffers from conflicts but the other does not, the hybrid predictor can select the well-behaved component.

An MAs predictor looks almost like a conventional two-level predictor, but by alloying local and global history, provides pseudo-hybrid functionality. This section examines the differences among an MAs predictor, a two-component hybrid predictor with dynamic selection, and a two-component predictor with static selection. The dynamic selector is itself a two-level predictor using global history [11].

### 4.5.1 Static vs. Dynamic Choice

The wrong-history category from the taxonomy only indicates whether programs as a whole benefit from having both types of history available. That data does not show whether branch sites within a program have different needs but particular branch sites consistently use the same type of history, or whether particular branch sites dynamically vary the type of history they use. In the latter case, static hybrid suffers. It must assign each branch site to one or the other component, and therefore mispredicts when that branch needs to use the other. Static selection also suffers when the training data is not sufficiently representative. Dynamic hybrid can learn how to steer a branch back and forth, and MAs also deals well with such behavior.

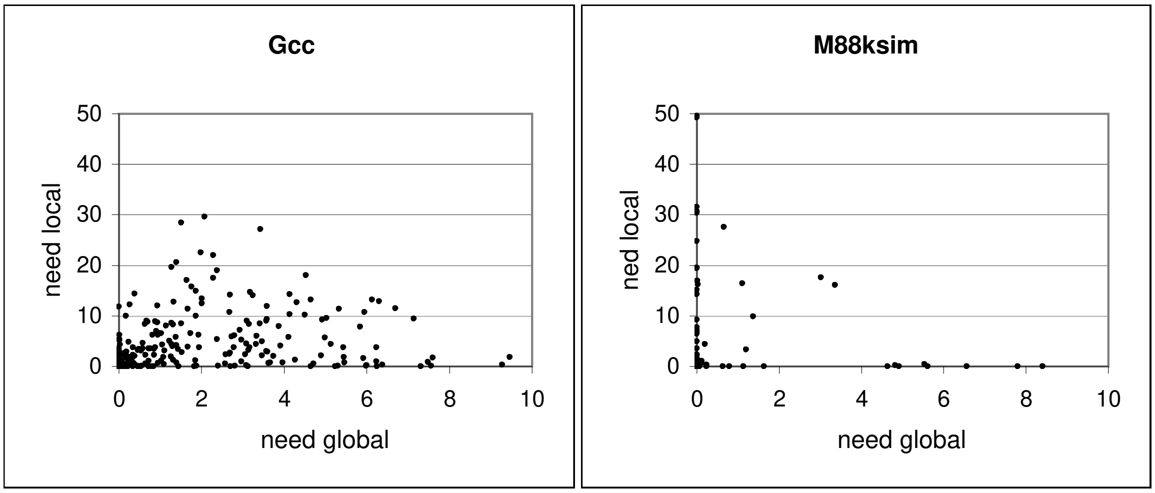


Figure 4.13: X-Y scatter plots showing, for each branch site, the percentage of time that branch needs a global predictor or a local predictor.

A better understanding of how to design hybrid predictors requires actually measuring how often wrong-history mispredictions occur at each branch site. Figure 4.13

shows X-Y scatter graphs for an 8 Kbit predictor with an interference-free BHT for *gcc* and *m88ksim*. Each dot represents one branch site; only branches which executed 100,000 times or more are plotted. The graph's x-coordinate gives the percentage of time that branch needs a global predictor; its y-coordinate gives the percentage of time that branch needs a local predictor. If branches consistently use one component, they will lie on one or the other axis. The more mass that lies in the middle of the graph, the more of a disadvantage static-hybrid prediction should experience. *Gcc* has many branches which use both local and global history; *m88ksim* fewer. In both cases, a substantial amount of mass lies at the origin, representing strongly-biased branches and other branches that are easily predicted by either structure (in the absence of interference). Branches that are mispredicted by both structures also lie at the origin.

### 4.5.2 Performance Comparison

Here only the largest of the sizes discussed so far, 64 Kbits, is considered. High-performance processors are likely to devote a substantial amount of resources toward obtaining good branch prediction accuracy. Furthermore, as shown in the next section, at smaller sizes like 8 Kbits, a hybrid organization's components are so small that even a stand-alone GAs or PAs is sometimes better.

As before, to get a fair comparison, the best configuration for each organization must be compared. For dynamic hybrid, I started with the components as close as possible to those used in Section 4.4.2, but added a selector. I then explored a wide range of configurations surrounding these starting points. The best configuration for a particular application varied widely, but for most programs, the best allocation of resources among the components gives more area to the global history component than to the selector or the local history component.

For static hybrid, I again started with the best components from Section 4.4.2, but deemed that the two components should use the same number of bits. This permitted a smaller search for the best configuration. The static selector was both trained and simulated using the same input, in order to eliminate unrepresentative training data as a problem. Note that this favors static selection; a more fair approach would cross-train, using different inputs for training and simulation. Yet even with this advantage, static selection is the worst of the choices.

| | GAs | | PAs | | | selector | |
|---|---|---|---|---|---|---|---|
| | index | PHT | index | BHT | PHT | index | PHT |
| Dynamic | 7g, 7a | 16K entries | 8p, 4a | 1K entries | 4K entries | 6g, 7a | 8K entries |
| Static | 7g, 7a | 16K entries | 13p, 0a | 1K entries | 8K entries | na | |

Table 4.7: Predictor configurations used for 64 Kbit dynamic-selection and static-selection hybrid predictors.

Figures 4.14 gives performance figures for the three choices, and Table 4.8 shows how much better or worse MAs's misprediction rate is. The hybrid configurations used for these results appear in Table 4.7. The MAs configurations were taken directly from Section 4.4.2.

Static-hybrid is worse than dynamic-hybrid except for *go*, where the reduced interference in the static organization is probably the dominant effect. These results differ from those reported by Grunwald *et al.* [30], which show that static selection has a significant advantage for smaller predictors, where it can substantially reduce interference compared to dynamic selection. For larger predictors, they find no difference between the two hybrid designs. The likely basis for the disagreement between their results and those presented here is simulation methodology. Their work does not describe the instruction set, compiler, or optimization level used to compile their

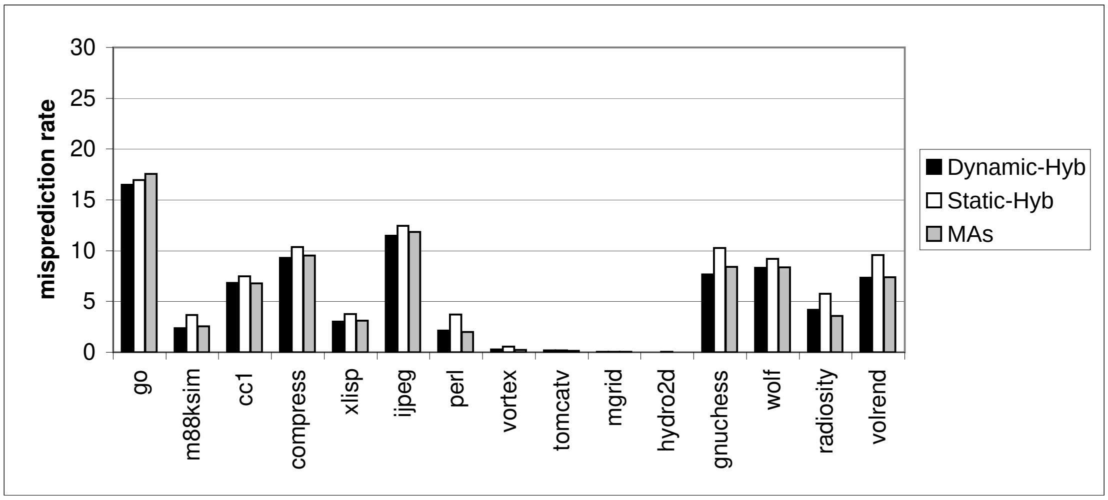


Figure 4.14: Relative performance of dynamic hybrid, static hybrid, and MAs for 64 Kbits total area.

| | Dynamic | Static |
|---|---|---|
| go | -6.24 | -3.35 |
| m88ksim | -8.18 | 29.99 |
| gcc | 0.54 | 9.35 |
| compress | -2.55 | 7.86 |
| xlisp | -3.14 | 17.48 |
| ijpeg | -3.05 | 4.86 |
| perl | 6.66 | 46.19 |
| vortex | 6.92 | 54.85 |
| tomcatv | 3.01 | 17.86 |
| mgrid | -5.36 | 13.24 |
| hydro2d | -4.55 | 23.33 |
| gnuchess | -9.25 | 18.46 |
| wolf | -0.76 | 8.98 |
| radiosity | 14.53 | 37.70 |
| volrend | -0.35 | 22.94 |

Table 4.8: Percentage of mispredictions that a 64 Kbit MAs predictor removes compared to dynamic and static hybrid predictors of the same size.

benchmarks. They also may not warm up the predictor state as this work does. That penalizes the dynamic selector if the simulation lengths are short, as theirs are—the dynamic selector takes some time to learn branches' behavior. This is the most likely source of the discrepancy.

If correct, my results suggest that individual branches are indeed alternating between using both global and local history. These suffer in the static organization, where such alternation is not possible. I have not measured the frequency of this alternation: some branches might change between history types just once.

Dynamic hybrid can mispredict even when one of the underlying components has the right outcome, if the selector *mis-chooses*. Mis-choices are a substantial fraction, even up to 50%, of mispredictions. When MAs outperforms dynamic selection, I have found that it usually does so because it correctly predicts some of the branches where mis-choices occur in the hybrid scheme. MAs usually outperforms static selection, but is roughly a tie compared to dynamic selection: sometimes superior, sometimes inferior, usually by small margins in both cases.

## 4.6 Alloyed History vs. Hybrid Prediction in Embedded Environments

Not all processors designed today can afford to devote a large area to the branch predictor. For example, power constraints may dictate a smaller chip size, and processors for embedded environments are generally both space- and power-constrained. These environments still require the best branch prediction available for the available size in order to maximize performance. Better branch prediction also reduces power consumption by reducing power expended on mis-speculated computation [61].

| | GAs | | PAs | | | selector | |
|---|---|---|---|---|---|---|---|
| | index | PHT | index | BHT | PHT | index | PHT |
| dynamic | 4g, 7a | 2K entries | 2p, 7a | 512 entries | 512 entries | 3g, 7a | 1K entries |
| static | 4g, 7a | 2K entries | 2p, 8a | 1k entries | 1k entries | na | |

Table 4.9: Predictor configurations used for 8 Kbit dynamic and static hybrid predictors.

For these reasons, this section revisits the 8 Kbit size and compares all the organizations discussed so far: GAs, PAs, dynamic and static hybrid, and MAs. Again, the best configurations are used. The 8 Kbit configurations for dynamic and static hybrid appear in Table 4.9. Figure 4.15 presents the results; it just adds the two hybrid predictors (the striped and stippled bars) to the data seen in Figure 4.11. Table 4.10 presents the fraction of mispredictions which MAs removes compared to other predictors.

Just as with 64 Kbits, dynamic hybrid usually outperforms static hybrid. *Go* and *gcc* are exceptions. But GAs and PAs sometimes outperform dynamic hybrid. The hybrid predictors must use such small components—dynamic hybrid must fit three structures into the available area—that these components are sometimes hobbled.

MAs can devote all its resources to a single structure while still providing both global and local history, so MAs performs consistently well. This chapter already showed that it does well against GAs and PAs. It also outperforms dynamic and static hybrid for every benchmark except *go*.

The alloyed-history organization, MAs, therefore shows excellent performance compared against other two-level predictors for a range of sizes and against hybrid predictors for smaller sizes. Compared to a large, 64 Kbit dynamic-hybrid predictor, MAs provides competitive performance. Furthermore, MAs performs consistently well over a wide range of benchmarks.

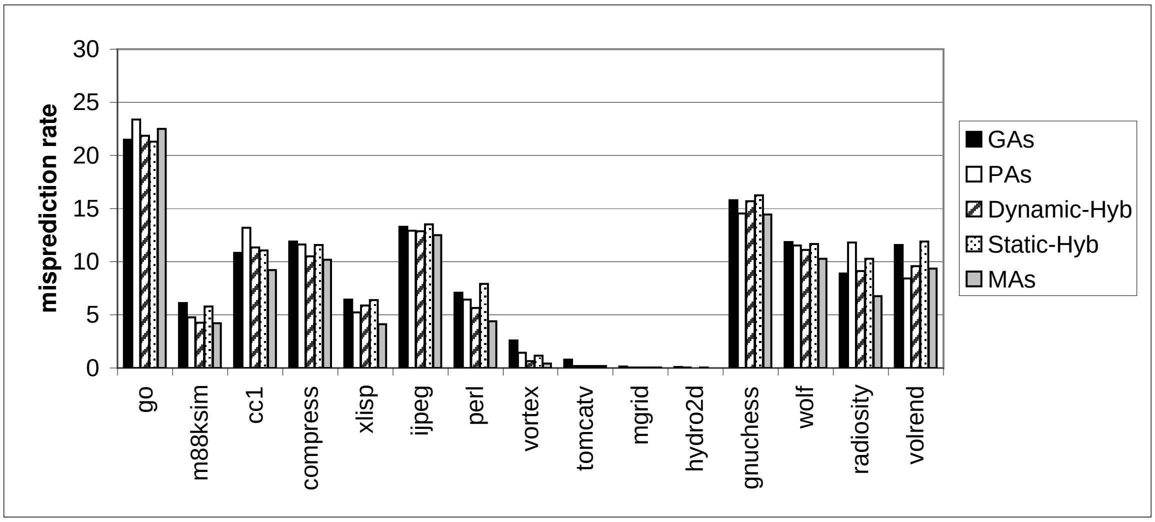


Figure 4.15: Relative performance of GAs, PAs, dynamic hybrid, static hybrid, and MAs for 8 Kbits total area.

| | Two-Level | | Hybrid | |
|---|---|---|---|---|
| | GAs | PAs | Dynamic | Static |
| go | -4.78 | -2.30 | -3.00 | -5.59 |
| m88ksim | 30.60 | 1.26 | 0.54 | 26.90 |
| gcc | 14.89 | 20.76 | 18.48 | 16.67 |
| compress | 14.15 | 12.32 | 2.97 | 12.03 |
| xlisp | 36.03 | 19.59 | 30.29 | 35.75 |
| ijpeg | 5.94 | 3.16 | 2.64 | 7.50 |
| perl | 37.81 | 20.66 | 22.20 | 44.56 |
| vortex | 84.42 | 40.36 | 37.77 | 65.16 |
| tomcatv | 78.47 | -4.82 | 6.95 | 12.56 |
| mgrid | 56.52 | -1.69 | 0.00 | 0.00 |
| hydro2d | 76.77 | 4.17 | 0.00 | 63.49 |
| gnuchess | 8.70 | -3.82 | 8.12 | 11.14 |
| wolf | 13.31 | 4.53 | 7.33 | 11.86 |
| radiosity | 24.27 | 42.83 | 26.20 | 34.49 |
| volrend | 19.32 | -10.94 | 2.70 | 21.55 |

Table 4.10: Percentage of mispredictions that MAs removes compared to other 8 Kbit predictors.

## 4.7 Related Work

Pan *et al.* [70] observed the tradeoff between including history and address bits in the PHT index, and introduced GAs. Yeh and Patt [115] further examined the balance between history and address bits in GAs and PAs, finding global history schemes suffer more from interference and are more strongly sensitive to both history length and number of PHTs (address length) than are local history schemes.

Sechrest *et al.* [82] also observed the value of sacrificing some history bits for address bits, especially for global history predictors. They further found that whether two shorter strings of history bits and address bits are concatenated, as in GAs/PAs, or full-index-length strings are XOR'd, as in *gshare* [64], has minimal effect on performance.

The above chapters explore the use of address bits to provide a kind of associativity in the PHT and therefore reduce destructive interference. Other researchers have described a variety of alternative techniques for reducing interference, the simplest of which is the aforementioned gshare. Sprangle *et al.* [94] measure the substantial harm caused by PHT interference and describe an *agree* predictor, which reinterprets the 2-bit counter's meaning as "agree with static prediction" or "disagree with static prediction". Michaud *et al.* [66] measure compulsory, capacity, and conflict aliasing and describe a *skewed* predictor, in which the branches simultaneously exist in multiple PHTs, and each PHT is indexed using a different hash function. A selector chooses which hash function to use. Lee *et al.* [57] observe that conflicting substreams may themselves by strongly biased, just biased in opposite directions. They describe a *bi-mode* predictor (not related to bimodal) that uses a meta-predictor to separate branch substreams of opposite bias into two PHTs. Because weakly biased branches still behave poorly under this approach, Eden and Mudge [16] follow this up with the

YAGS predictor, which tries to learn when branches disagree with the predicted bias. Chang *et al.* describe a somewhat different approach: *branch classification*, which identifies highly-biased (either taken or not-taken) branches, which may both cause and suffer from interference. They describe a mechanism to filter these branches out of the predictor reference stream.

Classification also reduces BHT interference. Other work to reduce BHT interference includes instruction reordering [8], and working set analysis [50], which identifies concurrently active branches and, using an approach analogous to register coloring, tries to ensure they use different BHT entries.

Yet other work has focused on characterizing why mispredictions happen in two-level predictors. Talcott *et al.* [96], Sprangle *et al.* [94], Young *et al.* [116], and Chang *et al.* [10] characterize PHT interference, and show that while both significant amounts of both constructive and destructive interference occur, the destructive interference consistently dominates. Young *et al.* also show, as does Nair [69], that using *path histories* improves performance: recording branch addresses instead of just the branch outcomes recorded by conventional branch histories can better separate branch substreams and reduce harmful interference.

Most recently, Evers *et al.* [21] focused on the correlation characteristics of branches, and found that global history strongly benefits a substantial fraction of branches within programs. Nevertheless, for a given prediction, most of the global history goes unused—adding to interference—and frequently the most useful branch outcomes have already been forced out of the global history. In a related vein, Sazeides and Smith explored how both data and control predictability propagates through the flow of a program in [81]. Emer and Gloy [19] used genetic-like mutations to grow predictors that merge information in a variety of ways.

Some branches within a program benefit more strongly from global history, while others benefit more strongly from local history. so McFarling suggests in [64] a structure that uses two components, with a bimodal, dynamic selector that chooses between them. Chang *et al.* [11] find that the most beneficial components are both two-level predictors: one using global history, the other, local history. They also show that a two-level organization using branch history as the selector outperforms a bimodal selector. Evers *et al.* [20] extend the two-component predictor by proposing a *multi-hybrid* predictor. This organization includes static-prediction components and a dynamic selector design that steers highly biased branches to these simple predictors. The multi-hybrid predictor also includes a bimodal predictor. Most recently, Grunwald *et al.*. [30] have proposed eliminating the selector in favor of a static selection mechanism, an approach included in this chapter's evaluations.

## 4.8 Conclusions and Future Work

A great deal of prior branch prediction work has focused on ways to improve two-level predictors using either local or global history, and some recent work has explored ways to combine multiple two-level predictors in one hybrid structure and how to choose which component to believe. This chapter presents a taxonomy of misprediction types to better understand how to further improve branch prediction design. The results show that in addition to the widely recognized problems of training time and interference in the pattern history table, *wrong-history* mispredictions are a substantial problem. These occur when a branch requires one type of history (global or local) but the predictor provides the other type. Not just programs as a whole, but also individual branches, suffer from wrong-history mispredictions. Hybrid predictors can attack these categories, especially wrong-history mispredictions, but the selection mechanism can be a further source of problems. In particular, static selection, which assigns branch sites to use one or the other component for the duration of the program, necessarily encounters mispredictions when individual branches suffer from wrong history.

Using these results, this work demonstrates the benefits of an *alloyed* predictor that merges local and global history together in a single PHT index. Although such an organization is an incremental change to existing two-level designs, it makes both types of history available all the time. This attacks wrong-history mispredictions, as well as a further category of mispredictions in which branches need *both* types of history simultaneously. Alloying can also reduce PHT interference by using a richer mixture of bits.

The proposed alloyed predictor, MAs, performs better than solo, two-level predictors of different sizes for most benchmarks. MAs also performs well against hybrid predictors, especially at smaller sizes where the components in a hybrid organization become too small. This is especially interesting for space-constrained environments like embedded processors, where better branch prediction not only helps performance, but reduces power consumption by reducing wasted computation on mis-speculated paths. Overall, these results show MAs to be an approach that yields robust results for a wide range of benchmarks and predictor sizes.

Finally, some preliminary data were presented, suggesting that a simple degree of configurability—allowing programs to dictate which bits comprise the predictor index—can improve the performance of all types of predictors, and MAs in particular.

There are several promising avenues for further research. Because MAs is a two-level organization like the GAs and PAs components currently considered in hybrid predictors, MAs might contribute the benefits of alloying when included in a hybrid predictor. In particular, it might work well in a static hybrid organization by helping individual branches that otherwise exhibit wrong-history mispredictions, while dispensing with the need for dynamic selection.

The results for hybrid prediction also suggest there still exists room for improving both MAs and hybrid prediction by better understanding when simpler two-level components fail and when selection mechanisms fail. Further refinements of branch misprediction categories will help this effort, too.

MAs is only one type of alloyed organization. Just as GAs and PAs can be improved by incorporating techniques for reducing interference, MAs can be, too. The mechanism for composing the alloyed index is a further interesting subject. Researchers have proposed techniques like XOR'ing history and address bits [64] that might benefit alloyed predictors as well.

# Chapter 5

# Speculative Update for Branch History

## 5.1 Introduction

Achieving the highest possible branch prediction accuracies is critical to good processor performance. Two-level-adaptive schemes [114] that use branch history have proven especially effective, because they correlate behavior among different branches. They also recognize more complex behavior patterns for a particular branch than simple up-down counters (like the table of two-bit counters described by Smith [90] and found in several recent processors [14, 67]). Two-level schemes now appear in at least two mainstream high-performance processors, the Pentium-Pro/Pentium-II [31] and the Alpha 21264 [32, 49]; in fact, the 21264 uses two different two-level predictors in conjunction with a selector to choose between the components.

The previous chapter developed a taxonomy of misprediction types, observed the importance of wrong-history mispredictions, and proposed alloyed branch history as a solution. Like most other branch prediction studies, it used instruction-level simulations that assume the correct result of one branch—regardless of whether it was mispredicted—is available to help make subsequent predictions. In a pipelined processor, however, the correct result is only determined when the branch is resolved. Between the time a branch is predicted and is resolved, 5, 10, or more cycles may elapse, during which time as many as 10 or 20 further branches may be fetched and predicted. If the processor only updates the predictor in the commit stage, predictions must therefore be made with "stale" state that lacks results from these branches that are still in flight. This particularly presents a problem for two-level-adaptive schemes, because these depend on accurate knowledge of previous branches' behavior, including those still in flight. This chapter therefore uses cycle-level simulations to explore a new category of mispredictions, those that result from non-speculative, or late, update.

Making branch results available without long delays is, however, feasible if the predictor can be speculatively updated with the predicted branch outcome instead of the resolved outcome. When predictions are correct, the speculative update causes no harm. Only in the case of a misprediction will a speculative update insert wrong information into the predictor, but the resulting damage can be repaired. This chapter describes the need for speculative update, the need for correcting the predictor's state after a misprediction, and mechanisms for accomplishing this.

When evaluating repair mechanisms, a further complication arises in those out-of-order processors that resolve branches as soon as they complete execution (*i.e.* the writeback stage), instead of at commit time. In this case, branches may resolve out of order, and any repair actions in the event of a misprediction must avoid corrupting or discarding state corresponding to prior, as-yet unresolved branches.

### 5.1.1 Related Work

This chapter presents work that also appears in [87]. Past research has shown the need for speculatively updating branch history registers. Yeh and Patt, in their 1992 paper comparing different two-level branch predictor organizations [114], state that speculative update improves prediction accuracy but do not describe mechanisms for repair after mispredictions. Talcott *et al.* [97], argue that older branch histories work just as well as newer ones, making speculative update unnecessary. Their evaluation approximates commit-time update by using histories that omit results from a fixed number of the most recent branches. Hao, Chang, and Patt [33] refute Talcott *et al.'s* results with cycle-by-cycle simulations showing that the number of outstanding branches varies, and this variability makes speculative history update necessary. Their results consider a GAp scheme, a theoretical predictor that maintains global history but provides each branch with its own PHT. Jourdan *et al.* [44] confirm these results for a single gshare predictor, and briefly suggest some mechanisms for repairing histories that have been corrupted by mispredictions, primarily focusing on global history predictors. This discussion primarily envisions storing necessary repair state in the instruction reorder buffer, as it draws on mechanisms for recovering register state that are described in [6]. The Jourdan paper also explores the benefits of speculative update for other branch predictor structures, and in particular shows that the PHT is insensitive to whether it is updated early or not. This insensitivity presumably results from the hysteresis in the PHT's 2-bit counters: a strongly-biased counter requires two changes to alter its prediction.

This work not only explores speculative update and associated repair mechanisms for global-history-based predictors in greater detail, but also explores speculative update for local-history (per-branch) based predictors, a topic which has previously received scant attention.

Although implementation and timing details about commercial processors are hard to come by, speculative update already appears in at least one announced microprocessor, the Alpha 21264 [49]. This processor uses two predictors, one based on global history and one based on local history, and dynamically chooses the prediction that is more likely to be correct. Only the global history uses speculative update.

### 5.1.2 Contributions

This chapter presents, for both global and local history schemes, a systematic, cycle-level evaluation using eleven SPEC95 benchmarks, of how much speculative history update can improve performance. These issues are explored as a function of branch predictor size and as a function of the number of branches allowed to be in flight at once. This detailed data substantially extends prior characterizations of speculative update's importance for global-history-based predictors [33, 44, 97], and to my knowledge is the first such published data for local-history-based predictors.

These results show that, as predictors grow larger, speculative update tends to matter less for global history predictors, but that it matters more for local history predictors. For both types of predictor, speculative update matters more as more branches are in flight simultaneously, but this effect plateaus for most integer benchmarks at a level of 4–8 in-flight branches.

This chapter also describes in detail several possible mechanisms for maintaining the required state for repairing branch history after mispredictions. Data are then presented that show how the decision to implement speculative update can affect the overall choice of branch-predictor type and configuration. In particular, providing speculative update for local history schemes trades off against combining predictors in a McFarling-style hybrid predictor [64].

### 5.1.3 Simulation Methodology

The results in this chapter are obtained from 50 million instruction long, cycle-level simulations of the SPECint95 benchmarks—just as in Chapter 3. Simulations use the HydraScalar simulator described in Chapter 2. Here, however, a more aggressive microarchitecture is simulated, with 8-way issue, a 128-entry instruction window, and 128 Kbyte caches. The configuration used appears in Table 5.1. The more aggressive configuration was simulated in order to glean the impact of speculative update and fixup on next-generation processors, where these mechanisms might appear.

The rest of this chapter is organized as follows. Section 5.2 discusses speculative-update effects in a global history predictor, and Section 5.3 discusses them in a local history predictor. Section 5.4 reports results for a hybrid predictor design, and Section 5.5 concludes the chapter.

## 5.2 Speculative Update for Global History

### 5.2.1 The Need for Speculative Update

Global history predictors, like GAg [114] and gshare [64], maintain a single history register that tracks the behavior of recent conditional branches. Each branch shifts its result into one end of this register, causing the oldest result in the register to be discarded. To make a prediction, a branch indexes the PHT (the table of saturating up-down counters) with this global history, and predicts taken/not-taken depending on the selected counter value. In some schemes, the index into the PHT also incorpo-

| Parameter | Value | Comments |
|---|---|---|
| **Processor core** | | |
| RUU size | 128 | Instruction window |
| LSQ size | 64 | Enforces load-store ordering |
| Decode latency | 4 cycles | Min time b/t fetch and issue |
| Fetch width | up to 8 instructions per cycle | Must be in same cache block |
| Decode width | up to 8 instructions per cycle | In-order |
| Issue width | up to 8 integer ops per cycle plus 2 FP ops per cycle | Out-of-order |
| Commit width | up to 8 instructions per cycle | In-order |
| Functional units | 8 ALU/logical (1), 4 branch/shift(1), 1 integer multiply/divide (12/20), 2 FP add (4), 2 FP multiply (4), 1 FP divide/sqrt (16/33) | Latency appears in parentheses |
| Memory ports | 3 loads/2 stores | any combination of 3 ops |
| **Branch prediction** | | |
| BTB | 2048-entry, 2-way | updated only if taken |
| Return-addr. stack | 32-entry | repaired after mis-speculation |
| Mispredict penalty | 7 cycles | 2 cycles for BTB-miss only |
| **Memory hierarchy** | | |
| L1 D-cache | 128 K, 2-way (LRU), 32 B blocks | 8 MSHRs, 1-cycle latency |
| L1 I-cache | 128 K, 2-way (LRU), 32 B blocks | 2 MSHRs, 1 cycle latency |
| L2 | unified, 8 M, 4-way (LRU), 32 B blocks | 4 MSHRs, 12-cycle latency |
| Memory | 200 cycle latency; 8 B/cycle | |
| L1→L2 bus | 1 transaction every 2 cycles | |
| L2→mem bus | 1 transaction every 8 cycles | |
| TLBs | 128 entry, fully assoc., 30-cycle miss latency | both I-TLB and D-TLB |

Table 5.1: HydraScalar configuration used in this chapter.

rates branch-address bits. Gshare, for example, XORs the history and the branch's address to produce an index, in order to reduce the likelihood that two branches compete for a PHT entry.

Tracking the history of all conditional branches together in one register like this permits the predictor to recognize correlation among branches. A program might, for example, have a sequence of statements like

```
B1:        if (A)
              ...
B2:        if (B)
              ...
B3:        if (A and B)
              ...
```

The third `if` is determined by the results of the prior two. Ideally, the outcomes of branches B1 and B2 appear in the global-history register by the time B3 must be predicted. Without speculative update, this often is not the case, so this section explores the need for speculative update of the global history register. (Regardless of when the history is updated, the PHT can be updated either speculatively or non-speculatively; as mentioned earlier, the hysteresis in the counters protects them from damage by speculative update, and the timing of the PHT's update has minimal impact on performance.)

A processor updating the predictor non-speculatively waits until instruction-commit to add branch results to the history register. The above example demonstrates the difficulty. Branch B3's outcome is predetermined by the results of B1 and B2. Unless aliasing interferes, a global predictor, once trained, should never mispredict B3, but to ensure a correct prediction for B3, the global history must contain the results of predictions B2 and B1. If the intervening basic blocks are sufficiently long, B2 and B1 indeed commit before B3 performs a prediction, and B3 predicts correctly. If the basic blocks are too short and the processor fetches B3 before resolving B1 and B2, the prediction of B3 finds a stale history that lacks B1's and B2's results. This delay does not necessarily cause a misprediction: B3 may be quite predictable on its own (error-checking code, for example), or well-correlated with branches prior to B1.

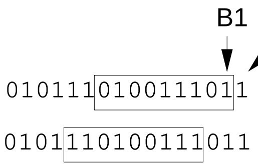


Figure 5.1: Two occurrences of branch B3 with the same actual history of preceding branch results. But different numbers of outstanding branches cause the history that appears in the global-history register (shown by the boxes) to vary.

Non-speculative update presents a second timing problem, described by Hao *et al.* [33]. Even if a particular branch can be well predicted using stale histories, out-of-order processors exhibit variability in the latency between two predictions. This occurs because dynamic events like cache misses, other mispredictions, and resource shortages cause the instruction window contents to vary.[1] On successive predictions of some branch, a particular bit in the global history register may therefore correspond to different branches. This means that even though the actual history of branch results prior to B3 may be the same each time B3 is encountered and identical predictions should result, different predictions may in fact occur. Different numbers of outstanding branches cause the global history register's contents to act as a moving window on the actual history. See Figure 5.1 for an example. Unless the predictor can train for all possible such windows —and this variability enhances the likelihood of aliasing—non-speculative update creates mispredictions even for predictable

[1]Even in-order processors exhibit variability if they use non-blocking caches.

branches.

The alternative is to update the history immediately after a prediction with speculative branch results. This guarantees that each prediction uses the most up-to-date history, solving the above problems. A new problem arises however, because every misprediction places an incorrect bit into the history register. Furthermore, updates from other branches on the wrong path also update the history register, adding further invalid bits. Unless some fixup mechanism repairs this corruption, the global history becomes unreliable and prediction accuracy suffers badly. Mispredictions can cascade, as one false history bit triggers subsequent mispredictions

### 5.2.2 Results

The prior discussion qualitatively illustrated the benefits of speculative update if implemented with a suitable fixup mechanism. The results in Figure 5.2 demonstrate quantitatively the importance of speculative update for global history predictors of various sizes, and the overwhelming need for repair mechanisms when doing speculative update. Each graph in Figure 5.2 shows, for a particular SPEC benchmark, the relative performance of (1) speculative history update with the described checkpointing of global histories, (2) commit-time history update, and (3) speculative update without any fixup. Results are presented for gshare predictors of different sizes; the x-axis shows global-history length, so the number of bits in the predictor is $2^{length}$ + a trivial number of bits for the history register. For these results, the processor allowed up to 32 in-flight branches.

**Speculative Update vs. Commit-Time Update.** The results are compelling: speculative update with fixup provides speedups of up to 31% (*go*) compared to commit-time update, with an average of 10% for SPECint using a 16-bit/64k-entry

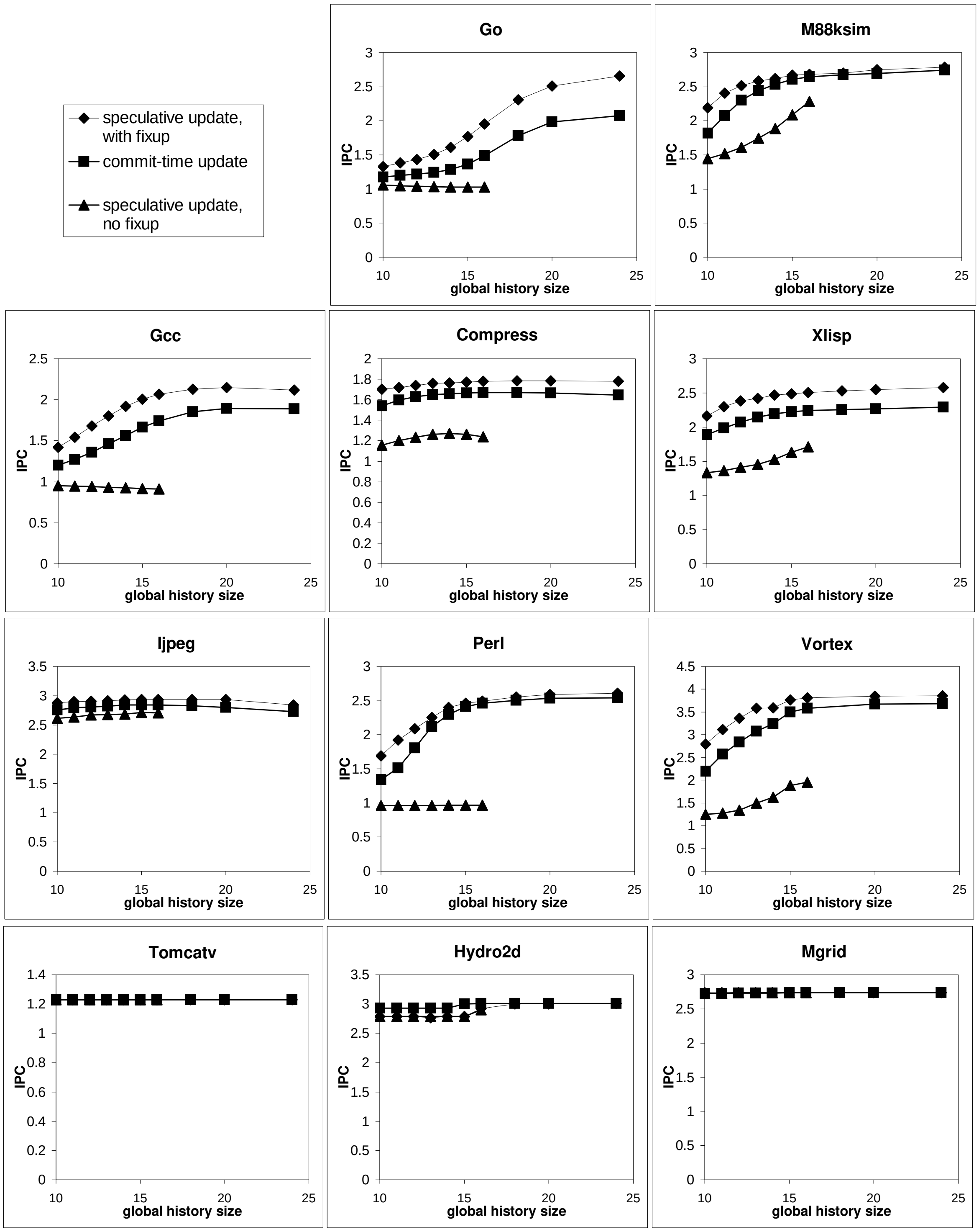


Figure 5.2: Performance of a gshare branch predictor as a function of size and history-update policy. The x-axis shows global-history length, and the PHT contains $2^{history}$ 2-bit counters. The curves coincide for tomcatv and mgrid. Results were obtained with 32 in-flight branches allowed.

gshare predictor. Among the floating-point benchmarks tested, speculative update has no benefit for *tomcatv* or *mgrid*. *Hydro2d*, on the other hand, benefits by 5-7% from speculative update for smaller predictors; omitting fixup for this benchmark cancels the benefits of speculative update but does not reduce performance below that of commit-time update.

The effect is less pronounced for *ijpeg* than might be expected with its poor prediction accuracy, but *ijpeg* program executes fewer branches, meaning that fewer branches are in flight at any one time, and with commit-time update the history is usually more up-to-date.

Similar tests were also performed for GAg and GAp predictors. In terms of overall performance, gshare outperforms GAg by a tiny margin for the largest predictors, and by a larger margin for smaller predictors. GAp outperforms both, again by a tiny margin for the largest predictors and by a larger margin for smaller predictors. Relative performance of speculative update with fixup, non-speculative update, and speculative update without fixup is the same, so only data for gshare are presented here.

**Speculative Update with and without Fixup.** In addition to the demonstrated benefit of speculative update, Figure 5.2 shows that speculative-update scheme requires fixup after detecting a misprediction. Otherwise most programs' performance suffers badly. The effect is so pronounced for many programs (*perl* and *gcc* provide especially dramatic examples) that speculative update without fixup is clearly not sensible—commit-time update is vastly better than speculative update without fixup—-and therefore simulation requirements have been reduced by limiting the number of data points gathered for no fixup.

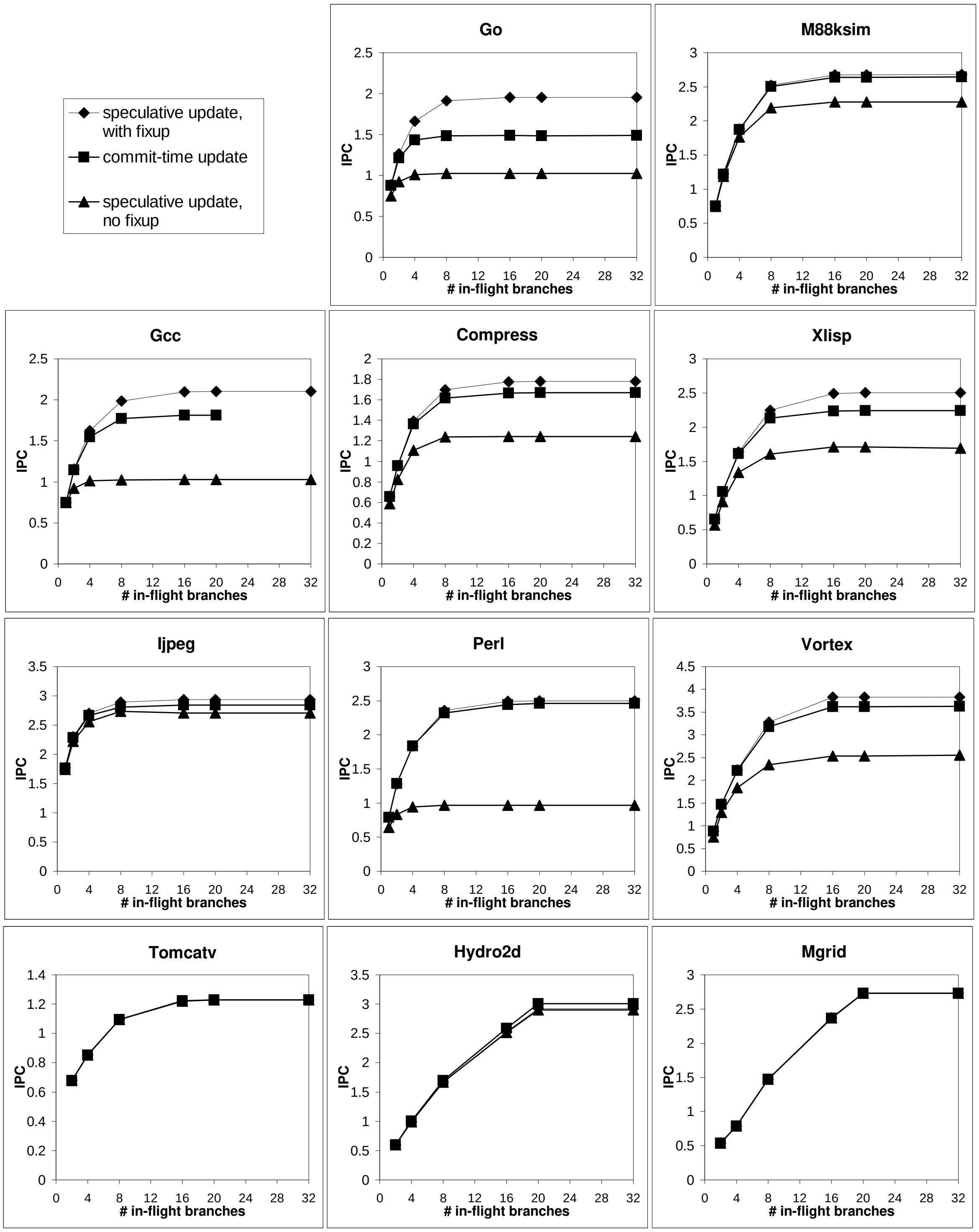


Figure 5.3: Performance of a gshare branch predictor as a function of the number of in-flight branches (both conditional and unconditional). All results in these graphs were obtained with a 16-bit gshare organization.

**Number of Outstanding Branches** Figure 5.3 shows the relationship among speculative update with fixup, commit-time update, and speculative update without fixup as the processor's in-flight-branch capacity varies. These simulations were done with a 16-bit global history but an infinite-sized PHT. At a value of 1, each branch commits before the next can be fetched, so speculative update has no effect. Organizations that permit a very small number of in-flight branches are of course untenable, since typical programs fetch many new branches before previous ones can traverse the pipeline, and so frequent stalls would result. These configurations are nevertheless included to more fully map out the shape of the curves.

The results show that, as more branches can be in flight simultaneously, commit-time global history becomes more out-of-date, making speculative update more beneficial. This trend flattens out at sufficiently large in-flight capacities of 4–8 branches for the integer programs. *Hydro2d* behaves similarly, although the effect is less pronounced, and the plateau occurs at 20 in-flight branches. As in the previous figure, *tomcatv* and *mgrid* are indifferent to the update timing.

### 5.2.3 Fixup Mechanisms

Given the benefits of speculative update and the need for history fixup after a misprediction, this section explores how to implement fixup.

Because the processor squashes all instructions after a branch misprediction, it is sufficient to find a way to restore the history that existed immediately before the misprediction occurred, and then update that history with the correct branch outcome (so that the mispredicted branch's correct outcome appears in the history). During the time between a misprediction and its detection, branches on the mispredicted path see corrupted history, but this is harmless because these wrong-path instructions are squashed. After the misprediction is detected and the history repaired, fetch is redirected to the correct path, and subsequent branches see correct history. If an earlier misprediction eventually resolves, it replaces the history register's contents with its own corrected outcome. In all this, PHT update can still take place at commit time.

Two basic techniques provide perfect fixup, in which every branch on a correct path sees the correct and up-to-date branch history, both mentioned in [44]. The first saves prior values of the global-history register, and maintains the most recent, speculative history in the register. The second only updates the register when the branch commits, and speculative history values are maintained separately, requiring the predictor to choose among two possible sources as the most up-to-date history.

**History-based.** In the first case, speculative updates modify the global-history register. Before the register's contents are modified, however, they are saved, for example at the tail of an *outstanding branch queue* or OBQ. This queue records prior history-register contents for all outstanding branches, in the order those branches were fetched. Predictions always read the global history register, and committing a branch merely discards the head of the OBQ. Fixup after a misprediction requires (1) a lookup in the OBQ to identify the misprediction's entry, (2) discarding subsequent entries, (3) restoring the global-history register's contents to their state just prior to the misprediction, and (4) adding the correct outcome of the mispredicted branch to the global-history register. Step 2 can be done in conjunction with steps 3 and 4. See Figure 5.4.

Step 4 can be done at prediction time instead, where it is not in the critical path and could be allowed to overlap into the next cycle. This entails saving in the OBQ not the global-history-register contents that existed just before the speculative update was made, but the corrected outcome that would have to be computed in step 4. This is achieved by shifting the global-history registers contents one position and shifting in the opposite of the branch's predicted outcome. (Because if the branch is eventually found to have mispredicted, the correct result will be the prediction's opposite.) See Figure 5.5.

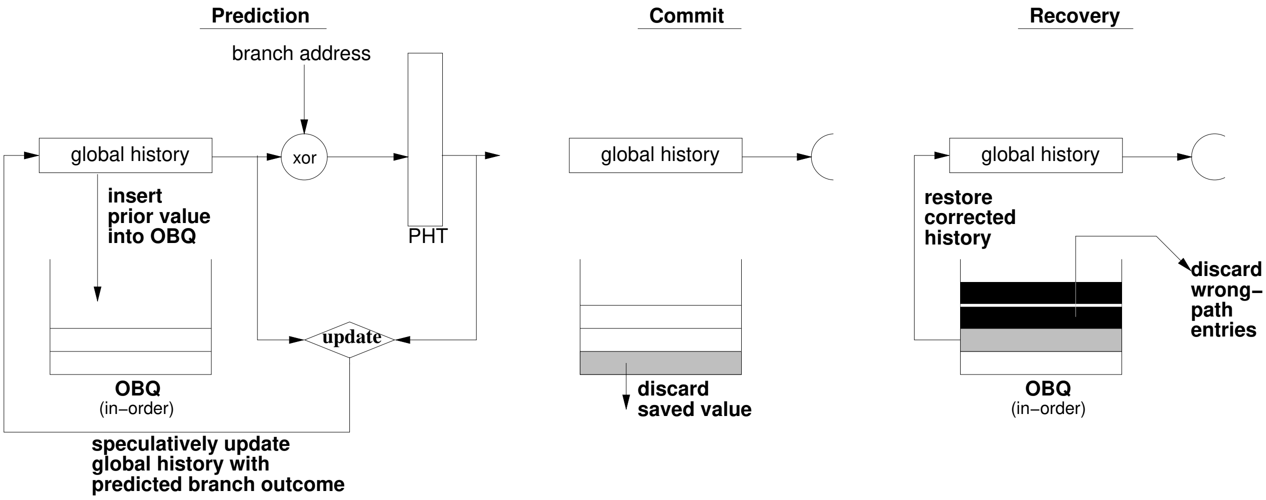


Figure 5.4: Checkpointing- or history-based fixup for a global history predictor. On a prediction, the current global history is saved in the OBQ, then updated with the predicted branch outcome. At commit, the saved global history can be discarded. When recovering from a misprediction, wrong-path saved histories are discarded, and the history that existed just before the misprediction is restored and updated with the correct branch outcome.

Because a particular branch might appear several times in the OBQ (e.g., a tight loop), the branch's instruction window or active-list entry should be tagged with its OBQ entry, so that step 1 above squashes the correct portion of the OBQ. Alternatively, the slightly longer global-history value can be directly saved in the instruction window, omitting the need for the OBQ. Jourdan *et al.* [44] describe this and also propose what is probably the most efficient alternative: using a wider global-history register which saves older history bits that would otherwise be discarded. Prediction now entails masking off the older bits; commit takes no action, and recovery shifts

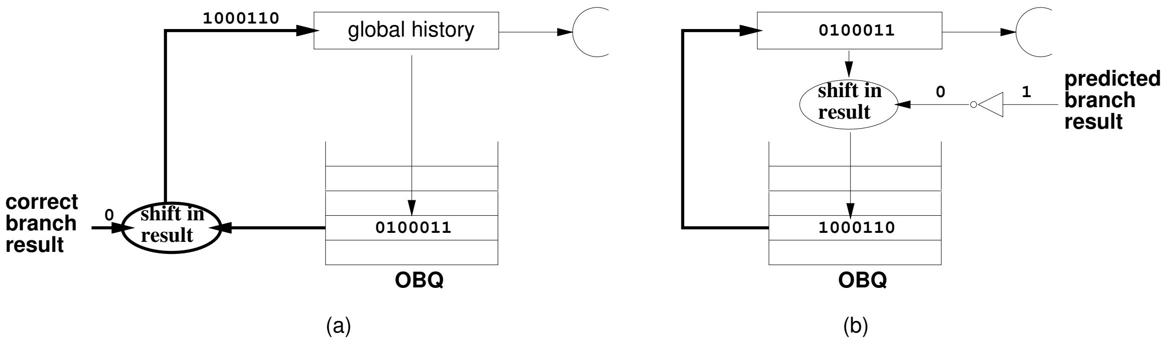


Figure 5.5: Two techniques for obtaining the corrected outcome at fixup.
(a) The OBQ saves histories, and shifts in the correct branch result at misprediction-recovery.
(b) The OBQ saves already-corrected histories; this works because the saved values are only used after a misprediction. Here the predicted direction was taken, but the branch was in fact not taken. In both (a) and (b), the bold lines at the left of the diagram indicate work at misprediction-recovery time.

the register contents to the right, discarding speculative bits, and also inverts the bit corresponding to the mispredicted branch. This shift-recovery scheme requires computing the right-shift distance (the number of speculative bits to discard) at misprediction-recovery time. This in turn requires tagging branches with a counter, and computing the difference between the mispredicted branch's counter value and the most recently assigned one.

As mentioned earlier, the Alpha 21264 performs speculative update for the global history component of its predictor. Fixup is achieved using an OBQ that stores prior values [49].

Regardless of the particular history-based technique used, the actions taken on the critical path for the common case, the prediction step, are unaffected. Misprediction recovery now requires restoring the global history register, but this can be done in parallel with other actions required during recovery.

**Future-based.** Instead of checkpointing the global history register in one of these ways, speculative updates can place the speculatively-updated history in the OBQ. This future-based mechanism uses the global history register only to store the committed history. The operation is slightly more involved than an OBQ that saves prior global histories and the associated alternatives discussed above, because a prediction might find the most up-to-date history in one of two places.

Making a prediction requires determining whether the OBQ contains any speculative histories; if so, the prediction uses the tail—the most recently created speculative history—of the OBQ, and if not, the prediction uses the global-history register. Committing a branch promotes the OBQ's head to the global-history register, as the new committed history. Fixup still requires knowing the mispredicted branch's corresponding OBQ entry, and discarding entries that follow the mispredicted branch, and the OBQ entry corresponding to the mispredicted branch must be updated with the correct outcome. This corrected result eventually propagates to the front of the queue and commits into the history register. As with the prior-history OBQ, the OBQ can be dispensed with by storing its contents in the instruction window.

Although the instruction window or active list provides an alternative location for maintaining ordered information about outstanding branches, it is further removed from the branch-history state than a dedicated structure. Using the instruction window requires extra space in that structure unless the window stores instruction results. In this case, if branches have no architectural results, the result field can store fixup information. Another location where history values might be stored is with shadow register maps, if the processor uses them to avoid associative lookups during register access.

An OBQ requires only a small amount of extra hardware. For each outstanding branch, the queue contains a history value. Because branches are tagged with OBQ indexes, no tags need to be saved in the queue. The queue's depth is determined by the maximum number of outstanding branches. If 20 branches can be outstanding and the global-history register is 12 bits long, as in the Alpha 21264 [49], this queue requires just 240 bits. The OBQ could be made shorter than the maximum number of outstanding branches if it stores prior history values. Although some state is lost, this state is only needed to recover from a misprediction, so no harm occurs if mispredictions do not occur in conjunction with dropped values. Since the OBQ is so inexpensive (and it can be dispensed with if history values are saved in the instruction window), this truncated version of the OBQ is not further explored.

The drawback to the future-based technique is that it may lengthen the critical path in the common case, the prediction step. Instead of just reading the global-history register and sending it to the XOR to be combined with the branch address, the predictor must see whether the OBQ has any entries (a simple Boolean test) and then choose whether the value sent to the XOR comes from the history register or from the OBQ (a multiplexor). Recovery after a misprediction is slightly simpler, on the other hand. Despite these timing considerations, the chief reason to consider future-based fixup for global history is that it dovetails well with the best mechanism for fixing up local history, as the next section describes.

## 5.3 Speculative Update for Local History

### 5.3.1 The Need for Speculative Update

Local-history-based predictors like PAg (Per-address, Adaptive, Global PHT) use a table of history registers (the BHT) instead of a global register. Typically, the branch address is used to index this table, producing a branch-specific history pattern that can then be used to index the PHT. Correlation among different branches does not occur in PAg unless branches happen to alias to the same history-table entry. This benefits branches that follow a reliable pattern, but that do not occur frequently enough for the global-history register to capture the pattern (*e.g.* non-innermost loop branches). Local-history-based prediction is of interest because it substantially outperforms global history prediction for some benchmarks [115], especially when mechanisms for reducing or eliminating aliasing in the BHT are assumed [50].

In a global history prediction scheme, flaws in the history register (stale state or corruption) potentially harm every subsequent branch: every branch reads the now-corrupted global-history register. In a local history scheme, on the other hand, a flaw in some BHT entry affects only the branches that read that particular entry.

Nevertheless, these results show that speculative update remains important for local history predictors. The impact is less than with global history predictors for the reason just mentioned. But multiple branches are almost always in flight at one time and multiple branches do sometimes alias to the same BHT entry. So with commit-time update, the value a branch reads from the BHT can be arbitrarily stale. As one simple example of how commit-time update can lead to mispredictions, consider a short loop with a short iteration count. Local prediction could ideally identify which iteration is the last. But if the branch-commit latency is too long, earlier iterations may not have updated the predictor by the time the last iteration of the branch is fetched. The loop should terminate, but the predictor's stale state does not yet reflect that, and a misprediction results.

When a local history predictor updates the BHT speculatively, fixup is required, just as with global history. Otherwise mispredicted branches place incorrect outcomes into the BHT, and other branches along the wrong path also put their results into their corresponding BHT entries. These wrong-path branches are squashed, but squashing does not undo the wrong-path updates to the BHT, leaving "extra" bits in the BHT from those squashed executions. In short, speculative update is not idempotent. These errors accumulate, and the entire BHT eventually becomes corrupted.

### 5.3.2 Results

For a subset of the benchmarks, Figure 5.6 shows the difference between a PAg predictor (1) with speculative update + fixup, (2) with commit-time update, and (3) with speculative update but no fixup, all as a function of BHT width. For these experiments, the BHT is infinitely deep, so every branch receives a dedicated entry. Figure 5.7 instead shows the comparison for different BHT depths, with a width of 20 history bits.

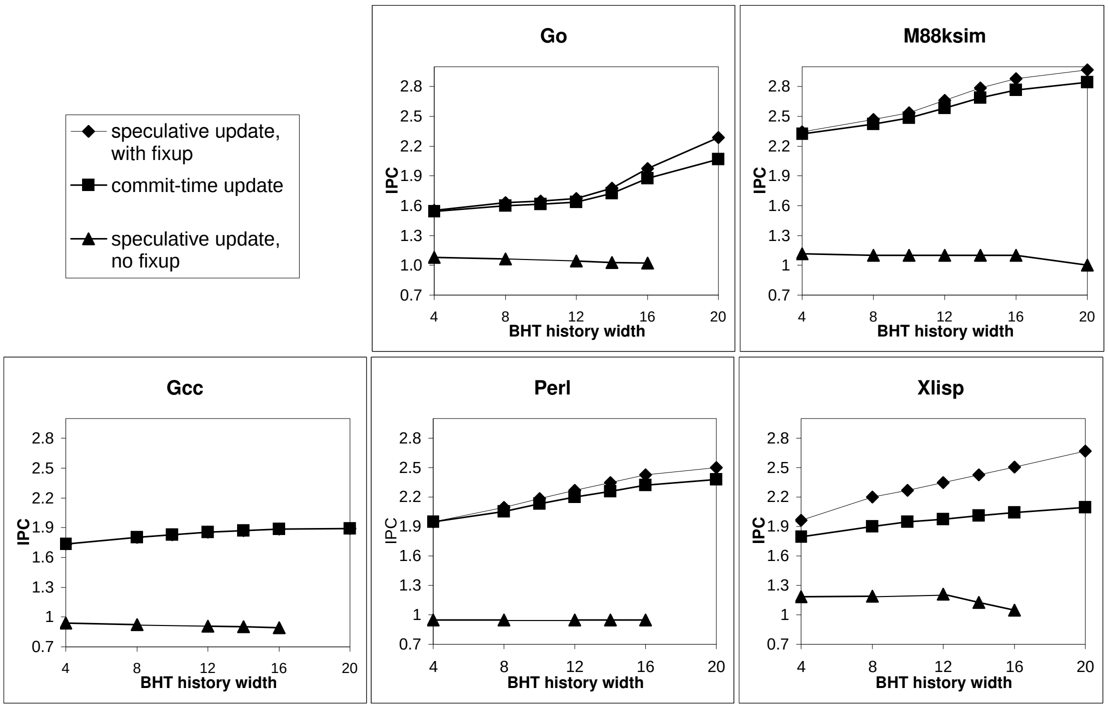


Figure 5.6: Performance of a PAg branch predictor as a function of BHT history width and history-update policy, for a BHT with infinitely many entries. The x-axis shows the history width, and the PHT contains $2^{history-width}$ counters.

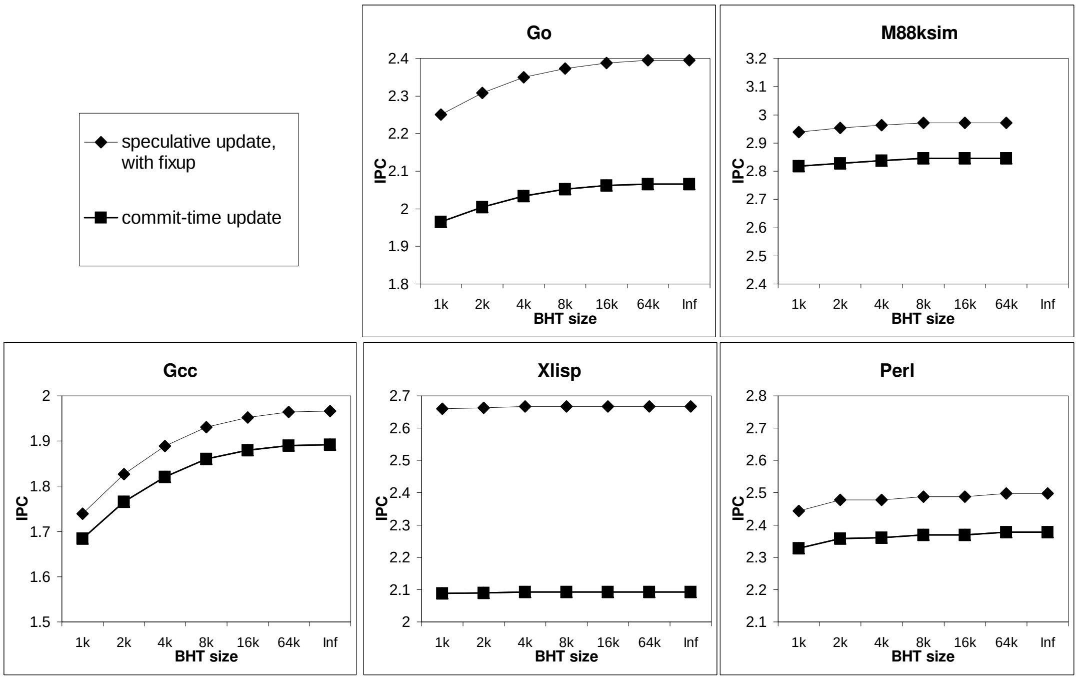


Figure 5.7: Performance of a PAg branch predictor as a function of BHT size and history-update policy, for a 20-bit-wide BHT. The x-axis shows the number of BHT entries, and the PHT contains 1-M 2-bit counters. The curves coincide for tomcatv, hydro2d, and mgrid, and are not shown. The trends are similar for 10-bit-wide PHTs, but as suggested by Figure 5.6, the gap between the curves is smaller.

The impact of speculative update is not particularly sensitive to the number of BHT entries (Figure 5.7), but is more sensitive to the history width stored in the BHT (Figure 5.6). As in the global case, speculative update performs terribly without fixup after mispredictions, and Figure 5.7 simply omits that curve.

In all cases except *xlisp*, the impact of speculative update compared to non-speculative is less for local history predictors than for global history. As mentioned earlier, a stale global history potentially affects all subsequent branches. But even with 20 or 30 outstanding branches and a local history predictor, only a subset of the BHT contains stale state. Branches using entries that don't correspond to these in-flight branches see up-to-date state.

Figure 5.8 shows the impact of different numbers of in-flight branches. As with

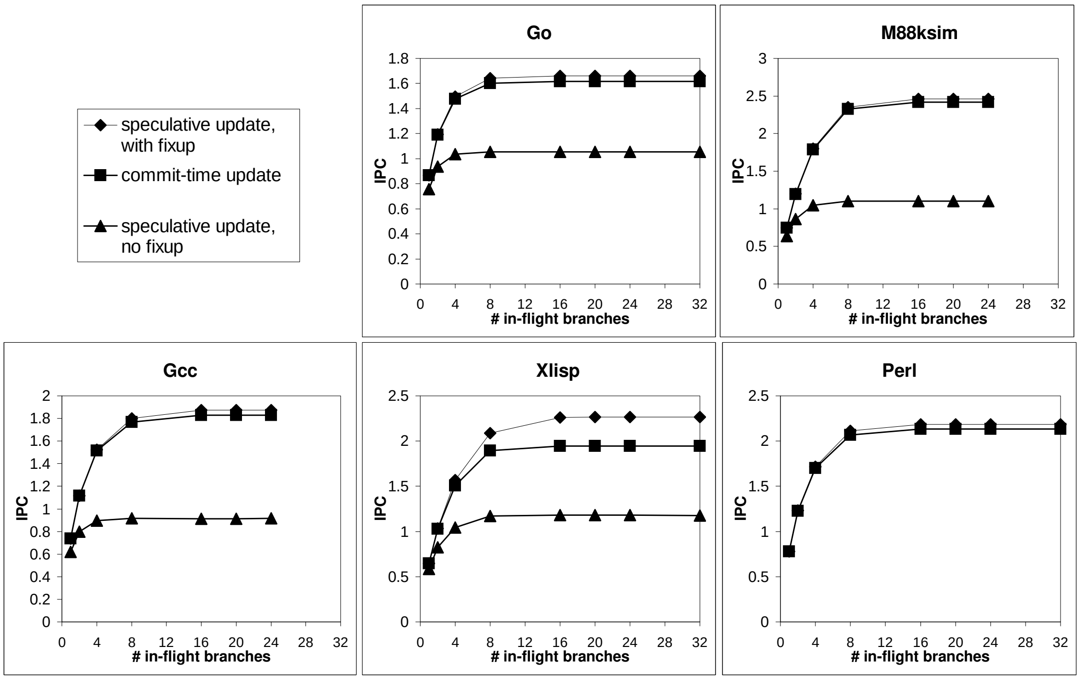


Figure 5.8: Performance of a PAg branch predictor as a function of the number of in-flight branches. The BHT is infinitely large and stores 10-bit histories (except for m88ksim, for which the graph shows data with a 1k-entry BHT). The speculative update, no fixup data were not available at submission time for perl.

the analogous global-history results, speculative update has more impact as the number of permitted in-flight branches increases. Without speculative update, the more branches in flight, the more out-of-date the history. As with global history, this trend reaches a plateau at 4–8 branches.

### 5.3.3 Fixup Mechanisms

Fixup for local-history-based predictors is more difficult. Unlike global history, where only a single register needs repair, potentially many BHT entries can be corrupted. Four repair schemes are possible, of which only one, the last, provides both time- and space-efficient fixup.

**Checkpointing.** First and most unreasonable is checkpointing, in which shadow copies of the BHT are maintained. Each shadow copy shows the BHT's state prior to a current in-flight branch. If that branch resolves without misprediction, it discards the associated checkpoint. In case of a misprediction, it restores the BHT and then adds the correct result for the branch in question. Both saving and restoring the BHT may take several cycles due to the quantity of information involved. This delays subsequent predictions unless some number of speculative bits are tolerated in the checkpoints, but that mostly defeats the purpose of checkpointing.

**History File.** A second possibility is a history file, whose size is a function of the number of outstanding branches and not of the entire BHT. The history-file mechanism resembles the favored scheme for global-history fixup, using an OBQ. It is essentially a selective form of checkpointing. Figure 5.9 shows its operation. Each prediction saves the contents of the selected BHT entry before updating that entry with the speculative prediction result. This save can be done in parallel with the PHT lookup, and otherwise prediction remains unchanged. When the branch commits, its saved entry resides at the OBQ's head and can be discarded. Unfortunately, recovering from a misprediction still entails a substantial amount of work. Beginning at the tail (which corresponds to the most recent speculative update to the BHT), BHT entries must be repaired one-by-one until the mispredicted branch's OBQ entry has been processed (it can be identified either with a prioritized associative lookup or with an OBQ tag on the branch instruction). The BHT now contains the same state as before the misprediction, and the mispredicted branch can update its BHT entry with its correct result. The mispredicted branch's OBQ entry should be saved, because it might already be on a wrong path, in which case subsequent fixups will need that information to perform further undo operations. This is an example of how out-of-order branch resolution affects the fixup mechanism.

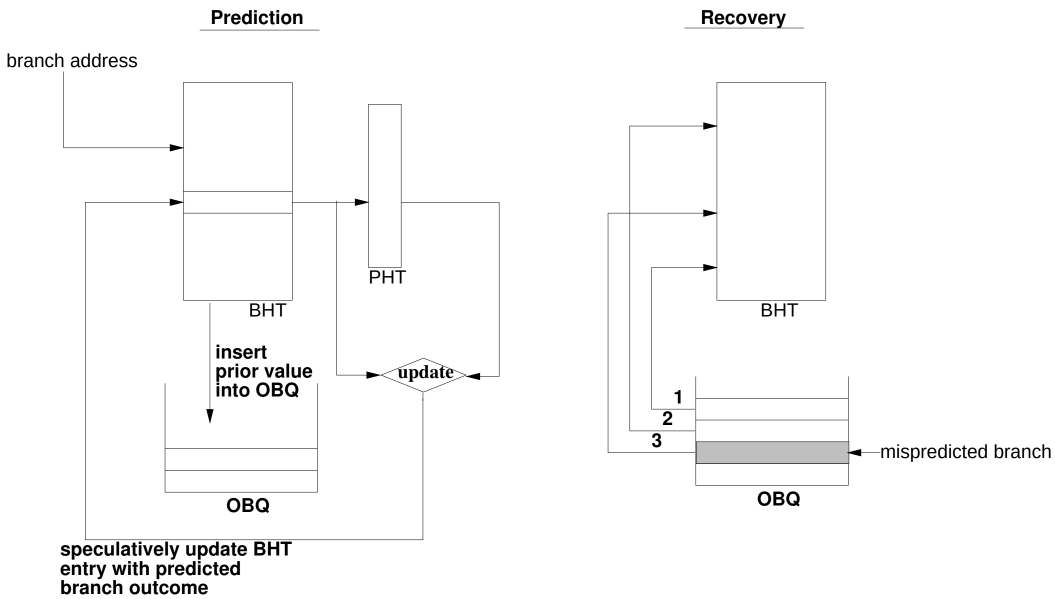


Figure 5.9: Repairing the BHT after a misprediction using a history file. The OBQ stores prior BHT entries that would otherwise have been overwritten. Recovery begins at the OBQ's tail and works forward to the mispredicted branch.

As with history-based fixup for global prediction, the critical path for the common case of making a prediction should not be affected. Unfortunately, the repair steps takes place one-by-one, because a particular location might require several undo steps.[2] Furthermore, the BHT has a limited number of ports. Even though mispredictions are not common-case events, they occur sufficiently often that the consequent delay significantly harms overall performance, as shown by Butler and Patt [6].

The history file's contents could be condensed by just storing the oldest bit that was discarded by the speculative update's left shift. Repairing an entry consists of a 1-place right shift and appending the discarded bit into the leftmost place. Alternatively, the saved history values can be maintained in the reorder buffer in place of the destination register contents, given the proper type of reorder buffer. Recovery

[2]Only the oldest value needs to be replaced, but the history values need to be maintained in the proper order.

becomes even more complex in terms of finding wrong-path branches among other reorder buffer contents.

**Future File.** Finally, instead of checkpointing the local history values, speculative updates can place the speculatively-updated history values not in the BHT, but in the OBQ, which then serves as a future file. The BHT thus always contains committed histories. As with future-based fixup in a global history predictor, the most up-to-date history can now reside in two places: prediction must determine whether to obtain the history value from the BHT or from the future file. Branch commit promotes a speculative history into the BHT. Except in an unordered future file, recovery becomes quite easy: future-file entries after the mispredicted branch are simply discarded, and the mispredicted branch's future-file entry has its last bit (the one corresponding to the misprediction) reversed. When the mispredicted branch commits, the correct value is promoted to the BHT.

There are several major variations on this scheme. Least attractive is an "unordered" future file, organized like the BHT, as in Figure 5.10. The file has as many entries as the BHT, and possibly several entries wide to accommodate multiple pending speculative updates. In other words, each BHT entry has a dedicated row in the future file. Finding a branch's entry is easy; as with the BHT, the index can be computed from the branch address, and neither associative lookup nor tag on the instruction are needed. This structure is potentially large, although the space can be reduced by only placing speculative bits in the future file, while the committed portion of the history is taken from the BHT. This, however, adds some further complexity to prediction, as the BHT's value must be shifted left by however many speculative bits reside in the future file and then merged with these bits.

A substantial problem with the unordered future file is misprediction recovery. Be-

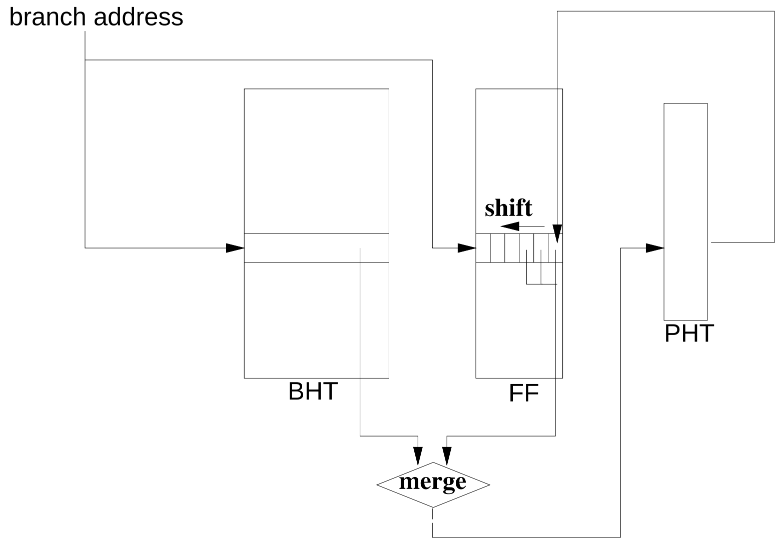


Figure 5.10: The prediction step using an "unordered" future file (FF). This diagram shows a future file that only stores speculative history bits, instead of entire history values. Here three bits are outstanding for the BHT entry in question. They are shifted left to accommodate the fourth bit which results from the prediction underway. Obtaining a PHT index requires left shifting the BHT entry and merging in the appropriate number of bits from the future file.

cause this organization maintains no ordering among speculative updates, squashing wrong-path information becomes difficult. This is another example of how out-of-order branch resolution affects the choice of fixup: in-order resolution would make this organization more attractive. Waiting until branch-commit or imposing in-order branch resolution ensures that all future-file contents are from the wrong path, but harms the performance of all branches, effectively lengthening the misprediction penalty [6]. Discarding the future file's contents at misprediction-detect discards needed correct-path state when the misprediction resolves before earlier, correct-path branches. Tagging entries with unique path identifiers, as in [1, 51], bloats the future file with extra bits and many comparators. Although space considerations prohibit presenting the data here, this unordered organization can, despite its drawbacks, allow speculative update to slightly outperform commit-time update. It still falls well short of the perfect fixup provided by an ordered scheme like the one discussed next.

A second choice is to store the speculative histories in the instruction reorder buffer given the proper type of instruction window, in lieu of the unused destination-register value. This imposes an ordering on the speculative values, and is similar to maintaining history state in the reorder buffer, as described above. When recovering from a misprediction, wrong-path values are discarded by the reorder-buffer cleanup process that discards the instructions following the misprediction. The mispredicted branch's speculative history is then corrected, so that the proper history will eventually be committed. The drawback to this choice is that the prediction step requires a prioritized associative lookup on the branch address in the reorder buffer, to find the most recent history for a particular branch.

**Future File with an OBQ.** The best choice is to maintain a future file using an OBQ. Speculative update values are kept in the OBQ. Prediction requires an associative lookup on the much smaller OBQ; recovery discards all entries after the mispredicted branch and corrects the mispredicted branch's entry. See Figure 5.11. As with global prediction, the OBQ must keep state for both conditional and unconditional branches.

I advocate the OBQ-based future file because the associative lookup is most likely less expensive than an associative lookup on the entire reorder buffer, and may well be as fast as indexing the much larger BHT. If this is true, since the BHT and OBQ lookups are done in parallel, the OBQ/future-file fixup mechanism only adds one multiplexor to the critical path for making a prediction. The OBQ's storage requirements are minimal. Recovery is fast and never loses information; although wrong-path branches may see incorrect history values, correct-path branches always see correct and completely up-to-date histories.

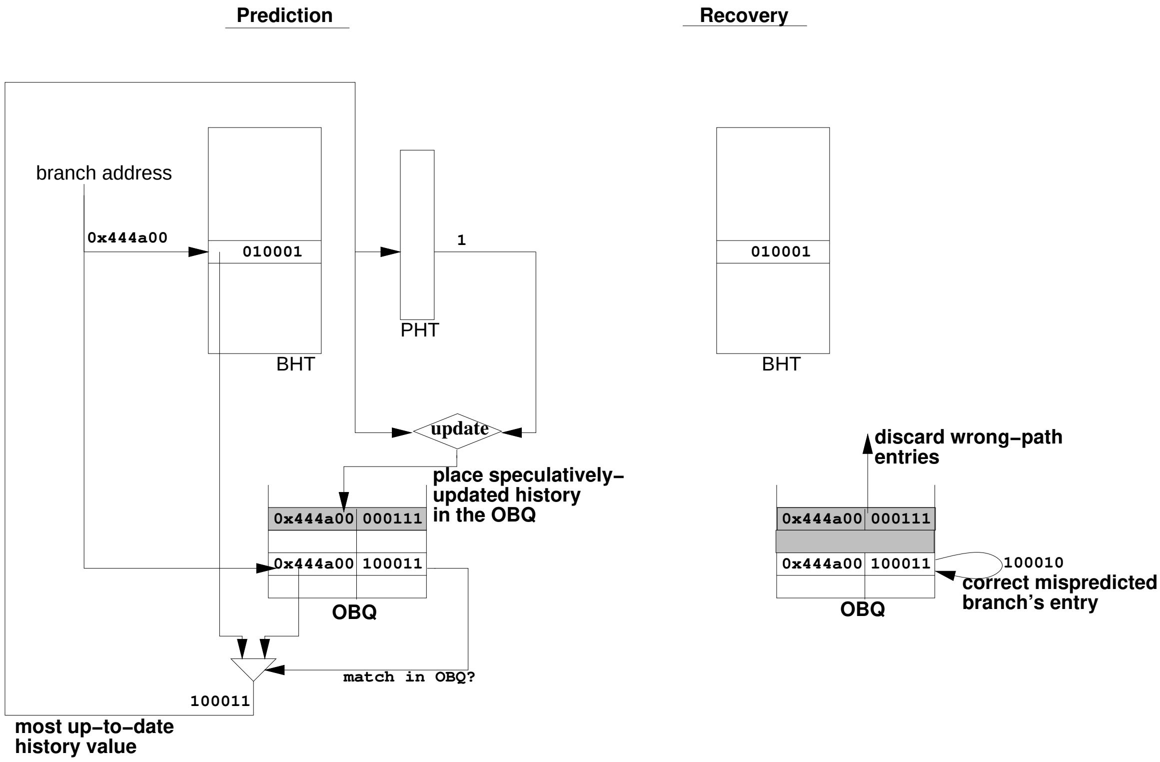


Figure 5.11: Repairing the BHT after a misprediction using an OBQ as a future file. The example shows a prediction (using the history value `100011` from the OBQ) of branch 0x4444a00. The new speculative history, `000111`, is added to the OBQ. But the first instance of 0x444a00 was a misprediction; subsequent OBQ entries are discarded, and the misprediction's entry is modified to reflect the correct result. The scheme operates similarly if the OBQ is omitted and the histories reside in the instruction reorder buffer.

## 5.4 Speculative Update with Hybrid Predictors

Previous sections have shown that global and local predictors both benefit from properly fixed-up speculative updates of history registers. The best performance for each benchmark is sometimes achieved with a global predictor, and sometimes with a local one. Because most programs contain some branches that perform better with a global history predictor and others that perform better with a local history predictor [21], it is tempting to combine them into a hybrid that will use, for each branch instance, whichever has recently been more accurate, as in a McFarling-style hybrid predictor [64]. The choice between local and global is made by a bank of 2-bit saturating counters, indexed by the global history register, as suggested by Chang, Hao, and Patt [11]. When one predictor is correct and the other incorrect, the counter value is either incremented or decremented, but does not wrap around (like the saturating 2-bit counters in the PHT).

| Finite | | | | | | |
|---|---|---|---|---|---|---|
| | | go | m88 | gcc | xlisp | perl |
| 1. | hybrid, spec-update | 1.61 | 2.75 | 1.88 | 2.51 | 2.47 |
| 2. | hybrid, late PAg update | 1.59 | 2.75 | 1.87 | 2.51 | 2.47 |
| 3. | gshare component only | 1.44 | 2.51 | 1.68 | 2.38 | 2.09 |
| 4. | PAg only, spec-update | 1.53 | 2.46 | 1.64 | 2.26 | 2.10 |
| 5. | PAg only, late update | 1.50 | 2.42 | 1.61 | 1.94 | 2.06 |
| 6. | equivalent-area gshare | 1.61 | 2.62 | 1.92 | 2.47 | 2.40 |
| Infinite | | | | | | |
| | | go | m88 | gcc | xlisp | perl |
| 1. | hybrid, spec-update | 2.86 | 3.05 | 2.24 | 2.77 | 2.70 |
| 2. | hybrid, late PAg update | 2.82 | 3.04 | 2.25 | 2.71 | 2.70 |
| 3. | gshare component only | 2.51 | 2.75 | 2.15 | 2.54 | 2.59 |
| 4. | PAg only, spec-update | 2.27 | 2.97 | 1.89 | 2.67 | 2.50 |
| 5. | PAg only, late update | 2.06 | 2.85 | 1.89 | 2.09 | 2.38 |
| 6. | equivalent-area gshare | na | na | na | na | na |

Table 5.2: Hybrid prediction.

The table shows that whether the local history component uses speculative update with fixup (line 1) or just uses commit-time ("late") update (line 2) has no effect on performance. The hybrid organization outperforms both its global component and local component alone (lines 3 and 4 respectively). Line 5 shows the local component's performance with just commit-time update. For the finite case, line 6 shows the performance of a gshare predictor with 14-bit history and 16k-entry PHT that uses approximately an equivalent number of bits as the hybrid predictor.

Because speculative update and suitable fixup are straightforward for global history, this section considers the impact of speculative update on the local history component of a hybrid predictor. The results suggest that, even though speculative update benefits a stand-alone local history predictor, it has minimal benefit for a local history component that is part of a hybrid predictor.

Two hybrid configurations are simulated. The first, loosely modeled after the Alpha 21264's hybrid configuration [32, 49], is a finite organization with 12 bits of global history. This makes both the selector table and the gshare PHT 4 K entries. The local component has a 1 K-entry × 10-bit BHT and a 1 K-entry PHT. The second is an "infinite" configuration. The global history is 20 bits wide, making the selector and the gshare PHT 1 M entries in length. The local component also uses 20 bits of history and its BHT has effectively an infinite number of entries (but Figure 5.7 says that 4 K is usually enough entries). In both cases, the global history (used by both the global history prediction component and by the selector) is always speculatively updated and fixed-up.

Table 5.2 shows the results. The hybrid predictor is, as expected, better than either of its components alone for both finite and infinite configurations (lines 1 and 2 vs. lines 3, 4, and 5). One component captures behavior that the other does not. Curiously, in both cases, *the hybrid predictor does not seem to need speculative update of its local component*—even for *xlisp*, for which local history benefits strongly from speculative update. Those particular branches that the local predictor handles better are apparently ones that are insensitive to update timing.

This suggests that hybrid organizations need only provide speculative update and the associated fixup mechanism for the global history component. The local history component can be updated at commit time. This is what the 21264 does [49]. The somewhat complex fixup—the future file—that local history requires can thus safely be omitted.

## 5.5 Conclusions and Future Work

This chapter has examined whether to speculatively update branch history in both global history and local history 2-level adaptive branch predictors, and described mechanisms for repairing the speculative history after mispredictions.

In particular, the data—obtained with cycle-level simulations of eleven SPEC95 benchmarks—show:

- Global-history-based predictors benefit strongly from speculative update.
- Repairing the speculatively-updated history after mispredictions is critically important, and practical hardware structures that implement perfect repair are described.
- Local-history-based predictors also benefit from speculative update, although less than do global predictors. The benefit increases as the history maintained in the BHT grows wider. Local prediction has received less attention in the literature, but with speculative update and sufficiently wide histories, a more hardware-efficient local scheme outperforms even huge gshare predictors for some programs.
- Repairing the history after mispredictions is again critical for local predictors using speculative update. The best scheme stores speculative histories in a queue of outstanding branches, and only places committed values in the BHT. This protects the BHT from corruption, and wrong-path contents can then easily be discarded. At prediction time, however, the processor must look in both the BHT and the outstanding-branch queue, requiring at least an extra multiplexor.

- For both global and local history predictors, the importance of speculative update increases with history width, and naturally also increases with the number of in-flight branches.

- Even though local history predictors benefit from speculative update (*xlisp* in particular), a hybrid predictor is apparently insensitive to whether its local history component is updated speculatively or not. This suggests that hybrid organizations can update the local component at commit time and avoid the hardware associated with speculative update and fixup.

The data in this chapter offer insights to hardware designers regarding the impact of fixup mechanisms in branch prediction hardware. The results also suggest that due consideration be given to local predictors in some cases. Finally, to researchers focused on simulation-based studies, the data also highlight the importance of careful simulation methodology in work involving branch predictors. In particular, researchers should use care in accounting for speculative-update effects in their simulators and in choosing appropriate hardware configurations when speculative-update effects are taken into account.

# Chapter 6

# Return-Address-Stack Performance and Repair

## 6.1 Introduction

The previous chapter showed the importance of early history update for two-level, conditional-branch predictors. It also described repair mechanisms to fix the history state in case a misprediction caused the early (speculative) update to insert incorrect information into the predictor. This chapter examines update timing for procedure-call *returns*, and shows that unless the return-address predictor is also updated early with suitable fixup mechanisms, return instructions become a large source of mispredictions. Note that return instructions are unconditional jump instructions, not conditional branches. This thesis is primarily devoted to improving accuracy of predicting branch directions, yet update timing affects not only branches, as described in the previous chapter, but also affects the accuracy of return-address instruction.

Procedure returns present the same problem as other indirect branches: because a procedure might be called from many different locations (consider `printf()`), the target of a particular return instruction varies. But extra information is available:

returns should pair up with corresponding subroutine calls. Since the return address is known at the time of the call (it is usually the next instruction after the call), a *return-address stack*[1] [46, 108] can match returns with corresponding calls. Like other prediction techniques, the stack's prediction is only a hint: the return instruction must still read the actual return-address register and commit. If the predicted return address is wrong, the corresponding mis-speculated computation must be squashed.

A simple stack, however, fails in the presence of speculative execution. The fetch engine predicts branch outcomes and speculatively fetches subsequent instructions: this may result in mis-speculated calls or returns that are later squashed. Calls and returns, however, update the stack in the fetch stage, while mispredictions are detected much later, in the writeback stage. Calls and returns on a wrong path therefore corrupt the stack. This becomes a substantial problem in current, high-performance processors that speculate aggressively. Without some repair mechanism to undo the effects of squashed instructions, the stack contents often do not correspond to the program's current call-return sequence, and return-address mispredictions result.

This chapter examines the behavior and performance impact of return-address stacks, concentrating on mechanisms for repairing or preventing corruption. Cycle-level simulation of the SPECint95 suite is used to evaluate several mechanisms, and to gauge sensitivity to stack size.

### 6.1.1 Related Work

The work presented in this chapter also appears in [85]. Otherwise, the literature discusses procedure returns infrequently. The most relevant work, by Jourdan *et al.* [44], evaluates repair mechanisms for the return-address stack as well as other branch

[1] This work uses "stack" and "return-address stack" interchangeably, and never speaks of the actual program stack in memory.

prediction structures. As this work does, they find stack repair to be necessary for effective return-address prediction. They evaluate a self-checkpointing mechanism that saves popped entries to avoid overwriting them with future mis-speculated pushes. Each stack entry, in addition to saving a return address, also contains a pointer to the next valid stack entry. Earlier work includes Kaeli and Emma's 1991 paper describing a two-stack return-address predictor [46], and Webb's 1988 paper [108].

Jacobsen, Rotenberg, and Smith discuss return-address prediction in the context of a processor organized around traces [39]. The trace mechanisms replace the return-address stack. Returns that do not appear at the end of a trace are followed in the same trace by instructions from the return site. These return-site instructions may or may not be correct, but replace the prediction that a return-address stack would provide. Traces which end in a return can be predicted using path-based next-trace prediction. This is particularly easy if the call appears in the same trace as the corresponding return. They also describe a *return history stack* which is not used to predict returns, but instead to cope with the loss of path history caused by subroutines. They observe that branch behavior after a subroutine call often correlates with behavior before the call. Unfortunately, subroutines of any substantial length fill up path history needed for next-trace prediction, replacing information about path history that precedes the subroutine call. The return history stack saves the pre-call history and restores it after subroutine returns.

Hily and Seznec evaluate return-address-stack performance in a simultaneously-multithreaded processor; because calls and returns from different threads can be interleaved, they find per-thread stacks are a necessity [35]. Chapter 7 considers return-address-stack performance in a multipath processor, which is in many ways like a multi-threaded processor.

Most work on control-flow prediction has focused on predicting directions of conditional branches [3, 7, 21, 57, 58, 90, 115, 116]. Attention is however turning toward other types of branches, as obtaining further improvements in conditional-branch accuracy becomes difficult. Researchers have recently begun focusing on general indirect jumps. Examining C++ programs, Calder and Grunwald observed that many indirect branches have a favored target. They proposed associating a two-bit counter with branch target buffer (BTB) entries: two consecutive instances of a particular target address must occur in order to change an indirect branch's target [9]. More recently, Chang, Hao, and Patt have proposed augmenting a BTB with branch history to select among various possible targets for a particular indirect branch. Instead of history bits indicating branch directions, their proposed history contains previous branch targets [12]. Driesen and Hölzle further examined history mechanisms for indirect-branch prediction [15]. Because returns are a special case of indirect branch, history mechanisms like these can potentially capture caller history well enough to distinguish among possible return targets. These general mechanisms, however, do not achieve the near-100% accuracies possible with a return-address stack. Because return-address stacks are so inexpensive and can be so effective, processors are likely to retain the return-address stack for the foreseeable future.

Most current high-performance processors include a return-address stack, but precise details about their management are rarely available. The DEC Alpha 21164 implements a 12-entry stack that can overflow and underflow [14]. The 21264 increases the size to 32 entries [32], and includes a (proprietary) stack-repair mechanism [18]. The Pentium MMX and Pentium II also implement a repair mechanism which uses valid bits to detect corrupted entries [25]. Cyrix recently patented a repair mechanism that preserves the top-of-stack pointer [65], similar to one of the mechanisms described here.

### 6.1.2 Contributions

This work closely examines the performance of return-address stacks in a simulated model of a contemporary processor, and finds that corruption caused by branch mispredictions is an important source of wasted cycles. A simple stack-repair mechanism is then described. It can remove essentially all speculatively-caused stack corruption, improving performance on the SPECint95 programs by up to 8.7%.

## 6.2 Design Issues

### 6.2.1 Basic Stack Operation

Figure 6.1 shows a simplified view of a modern fetch stage. Calls—either *jump-and-link* or *jump-and-link-register* instructions—push a return address onto the stack. Returns, instead of reading the BTB, receive a prediction by popping the stack. The most recent return jumps back to the most recent call site, and so forth; so long as calls and returns match up, returns are correctly predicted. The stack can be managed in a variety of ways, but is usually modeled as a circular LIFO buffer, pushed and popped during fetch. The stack can both overflow (overwriting the oldest entry) and underflow (returning invalid data from an already-popped entry). Section 6.3.4 briefly describes the role of overflow and underflow in stack performance.

Some architectures do not include specific return instructions, instead relying on an indirect-jump instruction. Returns can still be easily distinguished if they use a specific general register that other indirect jumps avoid, an easy task for the compiler. Some legacy code, if built with an older compiler that did not make such a distinction, might suffer from poor return-address-stack behavior unless corrected with a binary-rewriting tool.

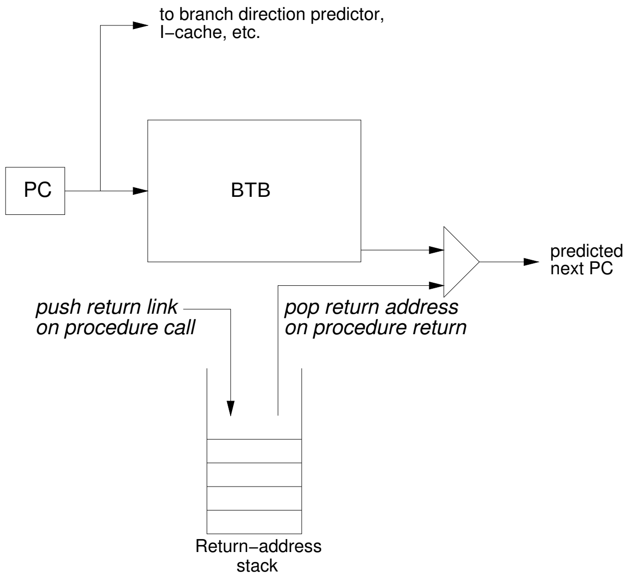


Figure 6.1: Basic operation of the return-address stack.

In a processor with no speculation, only overflow and unmatched call/return sequences corrupt the stack.[2] If the stack's prediction is treated as a hint, the stack needs no tags or valid bits. But a problem arises when speculative execution is permitted. After a branch misprediction, some number of cycles elapse before the branch condition resolves and the misprediction is detected. In the meantime, the processor speculatively fetches and executes further instructions which will be squashed. If any of the fetched instructions are calls or returns, they push or pop the return-address stack, because pushes and pops take place during fetch. These mis-speculated calls/returns will eventually be squashed, yet without additional state, the stack cannot be repaired—the stack just contains return addresses, with no way to determine which entries have been corrupted. A sequence of more mis-speculated pops than pushes or vice-versa throws the entire stack out of alignment, as Figure 6.2 illustrates. The `while` is incorrectly predicted to terminate. Subsequent returns and calls

[2] Such unmatched sequences might arise from events like `longjmp()/setjmp()` or exception handling; context switches also create the appearance of an unmatched sequence.

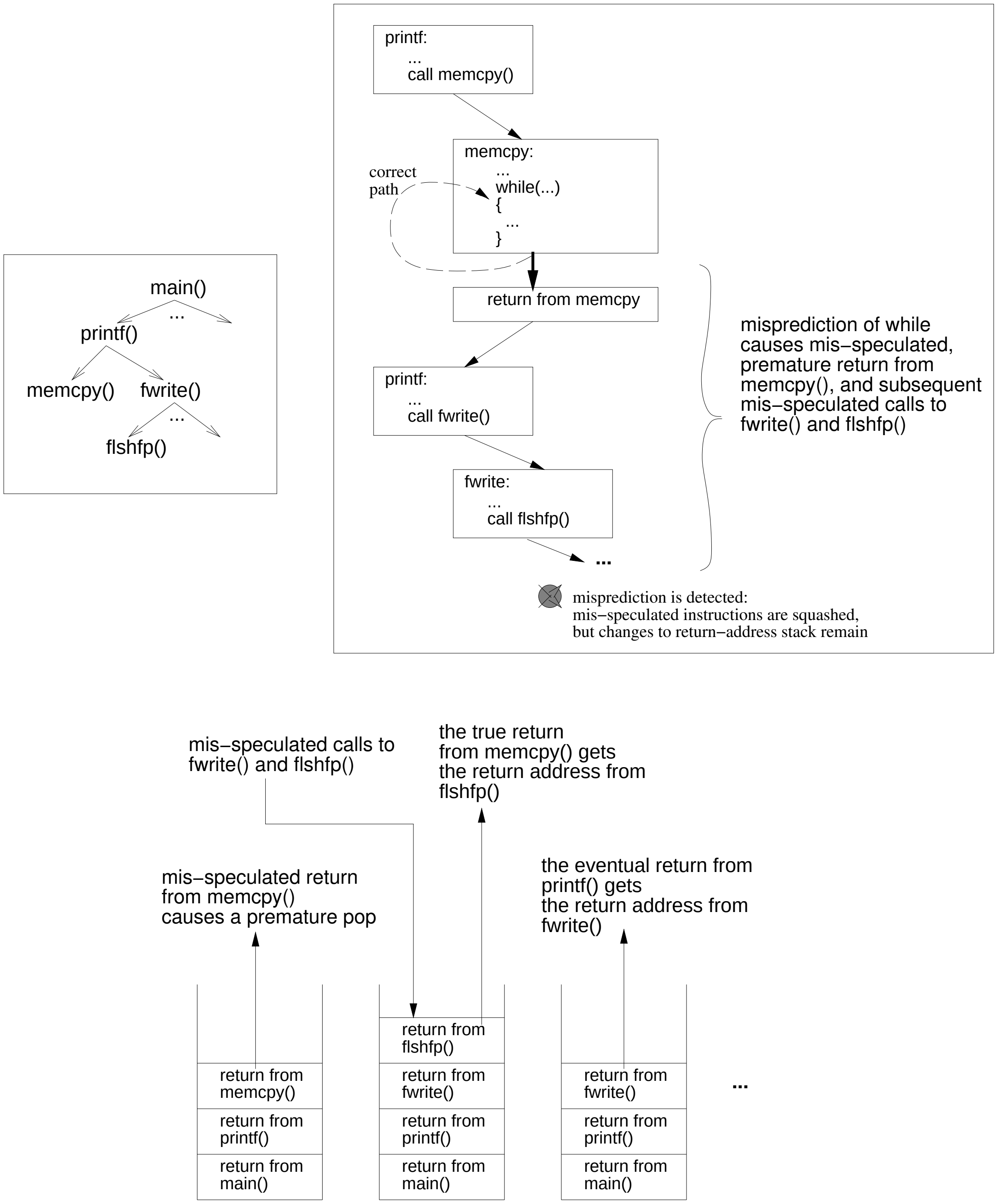


Figure 6.2: A case in which mis-speculation corrupts the stack.
A misprediction of the while in `memcpy()` causes a premature return and pop, and subsequent mis-speculated calls to `fwrite()` and `flshfp()`. Because the return-address stack is updated in fetch, these changes to the stack remain after the misprediction is caught. The correct return from `memcpy()` then pops the wrong return address, as do later function returns.

are mis-speculated and will eventually be squashed, but their effects remain in the stack even after the misprediction has been cleaned up. Correct returns then continue to mispredict, because each pop matches up with the wrong pushed value. If instead a matched number of mis-speculated pops and pushes occurs, the stack remains properly aligned, but with the wrong value in some entries. Note that because it takes some time for the branch predictor to be updated, the `while` might be mispredicted several times in a row, in which case this sequence of harmful effects happens each time.

The previous explanation assumed that pushes and pops take place during fetch. Updating the stack in the commit stage avoids corruption by mis-speculated instructions, but creates a different set of problems. Returns can be mispredicted even on a correct path, because an entire procedure can be in flight before the call at the beginning of the procedure gets a chance to commit and push its return address onto the stack. When the call has not yet pushed by the time the corresponding return is fetched, that return instruction reads a non-existent value from the stack. A further problem occurs when two or more returns occur in close succession. If the first one cannot pop the stack before the second one looks in the stack, the second one reads from the wrong location. The remaining discussion only considers repair mechanisms for stacks that are updated in the fetch stage.

### 6.2.2 Mis-speculation Repair Mechanisms

This work considers two simple repair mechanisms. Simply saving the current top-of-stack (TOS) pointer at the time of each branch prediction is sufficient to avoid losing stack alignment. It completely prevents corruption from a mis-speculated sequence of pops only or pushes only.

Saving the TOS pointer is reasonably cheap. Each time the processor speculates past a branch, it already saves some shadow state, such as the register-rename map, in case that branch is mispredicted. Saving the TOS pointer merely adds several bits per branch to this shadow state. The size of this shadow state is limited: in current processors, the shadow storage can accommodate at most a few in-flight branches—4 in the case of the MIPS R10000 [67], and 20 in the case of the 21264 [18].

Note that the shadow state may in fact be physically distributed: the shadow register maps near the register-mapping table, the shadow TOS pointers near the return-address stack, and so forth.

After a misprediction, the TOS pointer can be restored at the same time as the register map. No verification of individual stack entries is required. The stack itself remains simple: it still contains only the raw return addresses, without tags or valid bits.

A more aggressive scheme might also save the contents of the top stack entry along with the TOS pointer, requiring one additional (and much larger: 18–30 bits instead of just 3–5 bits) field in each shadow-state entry. After a misprediction, the TOS pointer and then the TOS contents are restored. Figure 6.3 shows a high-level schematic for this technique. Now a mis-speculated sequence of at least two pops followed by at least one push is necessary to corrupt the stack.

One can, of course, save an arbitrary number of return-address-stack entries this way; the extreme would be to checkpoint the entire return-address stack each time a branch is predicted. The results presented here provide data for full-stack checkpointing as an upper limit. The sophisticated scheme proposed by Jourdan, *et al.* [44], can have the effect of checkpointing the full stack, but requires a larger number of stack entries than the methods proposed here because it preserves popped entries. Each of these stack entries also requires more space, because in addition to storing the return

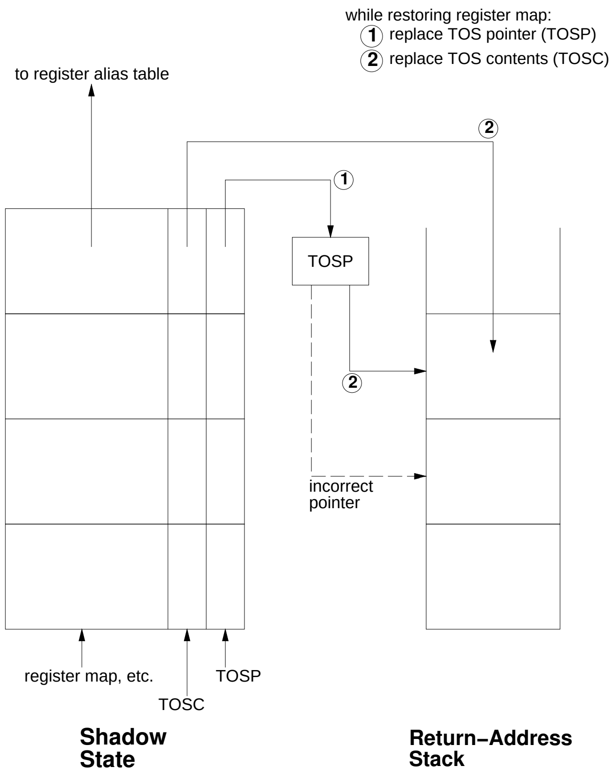


Figure 6.3: A high-level schematic of a return-address stack that restores both the TOS pointer and the TOS contents after detecting a misprediction.

address, a stack entry in this scheme also contains a pointer to the next entry that should be popped. On the other hand, this approach stores only two small pointers in the shadow state, the TOS pointer and the "NEXT" pointer (which points to the next free stack entry).

The cost of the TOS-pointer and TOS-contents ("pointer & data") repair scheme can easily be compared to the Jourdan *et al.* self-checkpointing scheme. Suppose the stack contains $n$ entries of 30 bits each (the lowest-order 2 bits can be discarded), and the shadow state supports up to $m$ in-flight branches. Note that the stack need not actually hold full 30-bit return-addresses, since text sizes are rarely so large.

If the stack does hold full 30-bit return addresses, the pointer & data scheme requires $30\,n$ bits for the stack, and $(30+\log n)\,m$ for the return-address-stack shadow state. The self-checkpointing scheme requires $(30 + \log n)\, n$ bits for the stack and $(2 \log n)\, m$ bits for the shadow state. With a 32-entry stack and 20-entry shadow state, like the 21264, the pointer & data scheme requires $960 + 700 = 1660$ bits, compared to $1120 + 200 = 1320$ bits for the self-checkpointing scheme—both quite small. The pointer & data approach also requires a wider bus between the stack and its shadow state (but these may be located close together): $30 + \log n$ bits vs. $2 \log n$ bits for the self-checkpointing approach's two pointers. Although this comparison has assumed the two approaches use the same size stack, the self-checkpointing scheme works poorly with very small stacks, because it does not have enough entries to store new return addresses while checkpointing older ones.

These proposed TOS-fixup mechanisms are simpler than tagging or using valid bits. Tagging must associate call and return addresses (and procedures might have many return instructions) and match tags on each access to determine whether to use the stack's prediction or the BTB's. Valid bits require identifiers for each in-flight branch; after a misprediction, these tags permit the processor to identify which stack entries have been corrupted.

## 6.3 Evaluation in a Conventional CPU

### 6.3.1 Simulation Technique

As before, *HydraScalar* is used to measure the performance of the SPECint benchmarks. Because the proposed changes are small, the configuration used for these experiments is not the aggressive model from the previous chapter, but instead, that described in Chapter 2 and used in Chapter 3. The same hybrid branch predictor with non-speculative history update, described in Chapter 3 on page 44 as the "baseline predictor", is used here.

Data are gathered for all eight SPECint benchmarks, using the same 50-million-instruction simulation windows described in Chapter 2. Note that, because these simulations do not include the effects of context switches, the results slightly overstate stack accuracy: context switches leave a corrupted stack when the process resumes. One way to avoid the resulting mispredictions is to save the stack as part of the context, but this should be unnecessary. Of chief importance in obtaining good return-address-stack performance is avoiding frequent corruption near the top of the stack, where most activity takes place. Since return instructions are so frequent compared to context switches, context switches should have negligible impact.

### 6.3.2 Repair Mechanisms

Let us begin by evaluating the repair mechanisms described earlier: *(i)* no fixup, *(ii)* TOS-pointer-only fixup, *(iii)* pointer-and-data fixup (restoring both the TOS pointer and the TOS contents), and *(iv)* full stack checkpointing. All four schemes update the return-address stack in the same way during fetch; they differ only in their repair mechanism. Table 6.1 compares the return-address-stack hit rates and speedups for these mechanisms with a 32-entry stack.

Without repair, return-address-stack hit rates are mostly in the 60–75% range, with an average of 71%. Aggressive fixup—restoring both the TOS pointer and the TOS contents—helps dramatically. It nearly eliminates return mispredictions, with corresponding speedups of 4.5–8.7% (average = 5.5%). Two benchmarks, *compress* and *ijpeg*, are exceptions and do not benefit from fixup. *Compress* executes procedure calls about as often as other benchmarks (see Table 2.3 in Chapter 2), and in fact has a fairly high conditional-branch misprediction rate; its call/return sequences are

| | Prediction Accuracy | | | | Speedup | | |
|---|---|---|---|---|---|---|---|
| | no fixup | pointer only | pointer & data | complete fixup | pointer only | pointer & data | complete fixup |
| go | 0.432 | 0.737 | 0.994 | 1.000 | 1.034 | 1.065 | 1.065 |
| m88ksim | 0.704 | 0.898 | 1.000 | 1.000 | 1.046 | 1.076 | 1.076 |
| gcc | 0.546 | 0.876 | 0.984 | 1.000 | 1.057 | 1.076 | 1.079 |
| compress | 0.993 | 0.997 | 1.000 | 1.000 | 1.001 | 1.002 | 1.002 |
| xlisp | 0.721 | 0.877 | 0.982 | 0.997 | 1.058 | 1.087 | 1.096 |
| ijpeg | 0.728 | 0.989 | 0.995 | 1.000 | 1.003 | 1.004 | 1.003 |
| perl | 0.664 | 0.928 | 0.963 | 0.990 | 1.078 | 1.087 | 1.100 |
| vortex | 0.901 | 0.993 | 0.999 | 1.000 | 1.043 | 1.045 | 1.046 |

Table 6.1: Return-address-stack fixup: prediction accuracy and speedup with a 32-entry stack. Speedups are normalized to the no-fixup case.

nevertheless well-behaved. *Ijpeg* has comparatively few calls to begin with, so the hit rate of only 72.8% barely hurts performance. For all these benchmarks, there is little need for saving more than the top entry of the stack contents: sequences of two or more pops followed by one or more pushes are rare.

A significant fraction of the benefit from pointer & data fixup can be obtained by restoring only the TOS pointer. Pointer-only fixup brings most of the hit rates up to about the 90% range (*go* is the exception), reducing mispredictions by 50–93%. Except for *compress* and *ijpeg*, which do not need any fixup, this yields speedups of 4.3–7.8%. The success of simple, pointer-only fixup indicates that mispredictions are often caught before a pop can be followed by a push on a particular mis-speculated path. The gap between pointer-only and pointer & data fixup then indicates the frequency of pop-followed-by-push sequences, and the small gap between pointer & data fixup and full stack checkpointing indicates the rarity of more complex sequences like two pops followed by a push.

These data indicate little need for more sophisticated mechanisms, like tagging, valid bits, or full-stack checkpointing.

### 6.3.3 Size

The previous data assumed a fixed-size stack of 32 entries; this section evaluates the contribution of return-address-stack size. Because fixup can make such a substantial difference, Tables 6.2 and 6.3 show stack hit rate and speedup for a stack with pointer-and-data fixup, while Tables 6.4 and 6.5 show similar data, but for a stack using no fixup. The column for size 0 in Tables 6.2 and 6.4 show the prediction success of the BTB: return addresses are found there just over half the time, on average.

With pointer-and-data repair, performance increases with stack size until about 4–8 entries. *Gcc* and *xlisp* benefit marginally from a further increase to 16 entries. Some programs, notably *go* and *compress*, get a large boost in performance from just a single stack entry. Others perform *worse* with a 1-entry stack than with no stack, which results from underflow: with a 1-entry stack, the stack only contains the most recent return address. A sequence of 2 returns pops an empty stack and almost certainly mispredicts the second return.

The presence of some sort of stack is important: adding a stack produces speedups of 6–15% (except for *ijpeg*, which has too few returns to matter). But without some repair mechanism, mis-speculation corrupts the stack so frequently that most of this gain cannot be realized; Tables 6.4 and 6.5 show that without fixup, performance is insensitive to the presence of a stack. Compared to a stack that omits fixup, only *compress* and *vortex* would be significantly penalized by just using the BTB.

### 6.3.4 Stack Overflow and Underflow

The simulations in this work have permitted the return-address stack to both over- and under-flow. *Overflow* occurs when a call is executed and the stack is full, either because of stack corruption or because there are more calls in progress than stack

| | 0 | 1 | 2 | 4 | 8 | 16 | 32 | 128 |
|---|---|---|---|---|---|---|---|---|
| go | 0.381 | 0.841 | 0.952 | 0.991 | 0.994 | 0.994 | 0.994 | 0.994 |
| m88ksim | 0.813 | 0.687 | 0.881 | 0.970 | 1.000 | 1.000 | 1.000 | 1.000 |
| gcc | 0.356 | 0.675 | 0.788 | 0.927 | 0.979 | 0.983 | 0.984 | 0.984 |
| compress | 0.510 | 0.976 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| xlisp | 0.683 | 0.741 | 0.834 | 0.923 | 0.956 | 0.976 | 0.982 | 0.984 |
| ijpeg | 0.793 | 0.911 | 0.971 | 0.996 | 0.998 | 0.998 | 0.998 | 0.998 |
| perl | 0.579 | 0.509 | 0.709 | 0.903 | 0.959 | 0.963 | 0.963 | 0.963 |
| vortex | 0.475 | 0.431 | 0.700 | 0.896 | 0.994 | 0.999 | 0.999 | 0.999 |

Table 6.2: Return-address-stack prediction accuracy as a function of stack size, with pointer-and-data fixup. For size 0, the BTB is the source of return-address predictions.

| | 0 | 1 | 2 | 4 | 8 | 16 | 32 | 128 |
|---|---|---|---|---|---|---|---|---|
| go | 1.000 | 1.047 | 1.061 | 1.066 | 1.066 | 1.066 | 1.066 | 1.066 |
| m88ksim | 1.000 | 0.987 | 1.035 | 1.057 | 1.059 | 1.059 | 1.059 | 1.059 |
| gcc | 1.000 | 1.051 | 1.071 | 1.097 | 1.108 | 1.109 | 1.109 | 1.109 |
| compress | 1.000 | 1.145 | 1.154 | 1.154 | 1.154 | 1.154 | 1.154 | 1.154 |
| xlisp | 1.000 | 1.011 | 1.042 | 1.082 | 1.093 | 1.101 | 1.104 | 1.105 |
| ijpeg | 1.000 | 1.002 | 1.002 | 1.002 | 1.002 | 1.003 | 1.003 | 1.003 |
| perl | 1.000 | 0.979 | 1.038 | 1.097 | 1.117 | 1.120 | 1.120 | 1.120 |
| vortex | 1.000 | 0.986 | 1.073 | 1.155 | 1.212 | 1.214 | 1.214 | 1.214 |

Table 6.3: Speedup as a function of return-address-stack size, with pointer-and-data fixup. Speedups are compared to having no stack, getting return addresses from the BTB.

| | 0 | 1 | 2 | 4 | 8 | 16 | 32 | 128 |
|---|---|---|---|---|---|---|---|---|
| go | 0.381 | 0.597 | 0.494 | 0.444 | 0.434 | 0.433 | 0.432 | 0.433 |
| m88ksim | 0.813 | 0.607 | 0.695 | 0.703 | 0.702 | 0.703 | 0.704 | 0.703 |
| gcc | 0.356 | 0.556 | 0.570 | 0.551 | 0.546 | 0.545 | 0.546 | 0.545 |
| compress | 0.510 | 0.974 | 0.993 | 0.993 | 0.993 | 0.993 | 0.993 | 0.993 |
| xlisp | 0.683 | 0.635 | 0.688 | 0.726 | 0.719 | 0.693 | 0.721 | 0.711 |
| ijpeg | 0.793 | 0.874 | 0.767 | 0.751 | 0.743 | 0.742 | 0.728 | 0.700 |
| perl | 0.579 | 0.471 | 0.588 | 0.663 | 0.662 | 0.663 | 0.664 | 0.663 |
| vortex | 0.475 | 0.426 | 0.682 | 0.849 | 0.901 | 0.901 | 0.901 | 0.901 |

Table 6.4: Return-address-stack prediction accuracy as a function of stack size, with no fixup. With size 0, the BTB is the source of return-address predictions.

| | 0 | 1 | 2 | 4 | 8 | 16 | 32 | 128 |
|---|---|---|---|---|---|---|---|---|
| go | 1.000 | 1.018 | 1.008 | 1.003 | 1.002 | 1.002 | 1.002 | 1.002 |
| m88ksim | 1.000 | 0.966 | 0.983 | 0.985 | 0.985 | 0.985 | 0.985 | 0.985 |
| gcc | 1.000 | 1.031 | 1.034 | 1.031 | 1.031 | 1.031 | 1.031 | 1.031 |
| compress | 1.000 | 1.144 | 1.151 | 1.151 | 1.151 | 1.151 | 1.151 | 1.151 |
| xlisp | 1.000 | 0.984 | 1.002 | 1.017 | 1.015 | 1.003 | 1.015 | 1.011 |
| ijpeg | 1.000 | 1.001 | 1.000 | 0.999 | 1.000 | 1.000 | 0.999 | 0.999 |
| perl | 1.000 | 0.968 | 1.006 | 1.027 | 1.026 | 1.027 | 1.026 | 1.027 |
| vortex | 1.000 | 0.984 | 1.066 | 1.134 | 1.162 | 1.162 | 1.162 | 1.162 |

Table 6.5: Speedup as a function of return-address-stack size, with no fixup. Speedups are compared to having no stack, getting return addresses from the BTB.

entries. The stack "wraps around," and as a result, the push overwrites the oldest stack entry. If a sequence of too many calls has caused overflow, a later return causes *underflow* by popping an empty stack, and the return receives an invalid result. This may be harmless if most call-return activity occurs near the top of the stack (for example, the overwritten entry might just be the call to `main()` from `crt0.o`). Underflow can also occur if mis-speculated pops, context switches, and the like destroy some values on the return-address stack and the stack later becomes prematurely empty.

An occupancy counter can detect instances of both overflow and underflow, but preventing overflow events is in fact undesirable. It saves the return address of the oldest calls in progress at the expense of newer information. Since the damage caused by overflow does not appear until a later underflow, the occupancy information can be used to detect when the stack is empty and in these cases the BTB can try to supply the return-address prediction. Unfortunately, in order to be useful, the occupancy-counter must be checkpointed with each branch prediction in the same manner as the TOS pointer and the TOS contents. Since over- and underflow are mainly a problem with small stacks, it may be better to just make the stack deeper.

## 6.4 Conclusions

This chapter evaluates return-address-stack design in light of potential corruption by mis-speculated instructions, a problem raised by Jourdan *et al.* [44]. Without a return-address stack, return addresses are found in the BTB only a little over half the time (Table 6.2). A well-designed stack can eliminate most or all of these return mispredictions, producing speedups of up to 15% compared to using only a BTB. But because modern processors speculate so aggressively, an effective return-address stack must incorporate a repair mechanism. Without such a mechanism, a stack provides little benefit for most SPECint programs.

Instead of trying to efficiently checkpoint the entire return-address-stack when speculating past a branch, a simple repair mechanism that restores the top-of-stack pointer and just one entry from the top of the stack turns out to eliminate almost all effects of stack corruption. Merely restoring the pointer performs fairly well, too.

This work only evaluated SPECint95 benchmarks, which are written in C. C++ programs or functional-language programs (Scheme, ML, etc.) sometimes have many short functions, in which case stack performance becomes even more important. Call/return and indirect-jump prediction for these environments are an important area of future work.

# Chapter 7

# Multipath Execution

## 7.1 Introduction

As the preceding chapters have shown, modern processors employ a variety of sophisticated branch prediction schemes to avoid delay penalties imposed by conditional-branch resolution. Yet some branches are intrinsically difficult to predict, and the mispredictions that remain cause serious disruptions in instruction flow as the processor wastes time following the wrong path until the misprediction is detected. Chapter 4 showed that the contribution of difficult-to-remove mispredictions is quite large. Worse yet, as issue widths increase and processor pipelines deepen, the misprediction penalty increases.

Since the delays caused by conditional-branch mispredictions remain a serious problem and yet some branches are intrinsically hard to predict, I investigate a different kind of remedy that does not rely on branch prediction: the simultaneous execution of *both* the taken and not-taken instruction sequences following a conditional branch, with cancelation of the one that turns out to be incorrect when the branch is finally resolved. Because additional branches are likely to be encountered before the first branch is resolved, this work considers the possibilities and potential benefit of executing more than two paths simultaneously. Each time *multipath execution* forks successfully, it eliminates a misprediction, although possibly with the expense of increased hardware contention.

Ideally, forking would happen only on mispredicted branches. A *confidence predictor* [1, 29, 38] attempts to evaluate the likelihood that a branch has been correctly predicted. Multipath execution uses confidence prediction to reduce hardware contention: instead of forking, the processor speculates conventionally past higher-confidence branches, following only the predicted path.

Multipath execution requires additional hardware throughout the processor, of course. The most affected section is the instruction fetch unit, which needs extra resources for each path. Since more instructions will be in flight at once, additional functional units, registers, cache ports, and so on are also needed, but the structure and control of this part of the machine are very much like contemporary multi-issue dynamic-execution processors.

### 7.1.1 Contributions

This work provides information for architects and designers of future processors who would like to include multipath capabilities. It is part of joint work with Pritpal S. Ahuja that appears in [1]. This chapter describes my contributions to the project. In particular, this chapter investigates the following:

- What changes need to be made to a conventional processor to support multipath execution?
- If hardware resources are abundant, how close can a multipath design come to the performance of perfect branch prediction?

- How should instruction cache bandwidth be apportioned among competing paths?

The results show that multipath execution can offer sizeable IPC improvements over more traditional single-path execution models: four paths yield speedups of up to 30% on future-generation processor configurations. For programs with very high accuracies which are unlikely to benefit from multipath execution, this work shows that performance does not decrease. Although four-path speculation would have a significant hardware cost, there are extensive overlaps with the hardware required by multithreading approaches such as simultaneous multithreading (SMT) [99], and combinations of multipath and multithreading make efficient use of such hardware [107].

The results also demonstrate that total instruction fetch bandwidth is a key lever on performance, and that heuristics to devote extra fetch resources to probably-correct execution paths can help further improve performance.

### 7.1.2 Related Work

Two recent papers are closely related to this work. Wallace, Calder, and Tullsen show significant speedups with a processor organization that combines SMT with multipath execution: thread contexts not used for program-level threads are used for multipath execution [107]. Klauser, Paithankar, and Grunwald describe Selective Eager Execution, a multipath organization similar to that described here, and report similar speedups [51]. A third paper by Kol and Ginosaur [52] examines an aggressively decoupled processor for a high-transistor-count processor that uses multipath execution to increase instruction throughput.

In earlier work, Heil and Smith used trace-driven simulation to study the performance of dual-path execution [34]. Tyson, Lick, and Farrens described how to use state from a two-level branch predictor's history table as a dynamic confidence predictor [102]. Mechanisms for dynamically computing branch prediction confidence were also investigated by Jacobsen, Rotenberg, and Smith [38], by Grunwald *et al.* [29], and by Pritpal Ahuja as part of this multipath project [1]. Still earlier, Uht and Sindagi [104] investigated *disjoint eager execution*, in which static probabilities were attached to each branch outcome, and forking decisions made in favor of the path having the highest likelihood. Wrong-path prefetching, which prefetches the first block on the unpredicted path of a branch, was explored by Pierce and Mudge [71] and can be considered a much simpler form of multipath execution. Machines from the late 1970s (the IBM 3033 and 3168) actually incorporated a form of wrong-path prefetching.

The remainder of this chapter is structured as follows. Section 7.2 discusses the proposed organization for multipath execution in more detail. Section 7.3 then describes the simulator and benchmarks. Section 7.4 explores the limits of the multipath idea, Section 7.5 examines fetch bandwidth, and Section 7.6 combines the previous ideas by looking at a realistic (though forward-looking) organization. Section 7.7 concludes the discussion.

## 7.2 Multipath Execution

At those conditional branches where it is difficult to predict the correct execution path, simultaneously executing both paths prevents losing cycles to misprediction and recovery. Each resulting path, or thread, may reach further branches; given sufficient hardware, control may fork once again. Branches which do not or cannot fork are predicted and speculated conventionally. Once a branch resolves, the wrong path and any of its child paths are squashed. The benefits from eliminating mispredictions in this way are tempered by the increased contention for resources as paths divide. This work proposes and evaluates mechanisms for implementing multipath execution, and for controlling the potential explosion of paths.

### 7.2.1 Differences from Conventional CPUs

The baseline hardware described in this chapter has three main differences from current high-performance CPUs:

**Confidence Prediction and Fork Control Unit.** At each conditional branch, deciding whether to fork an additional path depends on *(i)* whether there are any path resources currently free and *(ii)* whether the branch is likely to be mispredicted. Item *(i)* is easy to track, but item *(ii)* is more difficult. A simple policy would be to spawn a new path at every conditional branch as long as resources are available; this can be referred to as *naive* forking, and its performance results appear in Section 7.1.

More elaborate policies try to reduce hardware requirements by being more precise in determining (*i.e.,* guessing) when the branch will be mispredicted. For these policies, a dynamic confidence prediction unit is assumed to operate in parallel with the (more familiar) branch prediction unit. While the branch predictor returns its prediction of whether or not the branch will be taken, the confidence predictor returns its prediction of whether or not the branch predictor will be correct. One could philosophically consider the branch and confidence predictors to together constitute an even more elaborate branch predictor, but this study assumes traditional branch prediction hardware, and the existence of a separate confidence predictor. The upper bound on the performance of all such schemes is demonstrated by a strategy that forks precisely at the right time, on every branch misprediction (when path resources are available). This can be termed *omniscient* forking. Note that this can still perform worse than omniscient branch prediction: forking can create resource contention, and if mispredictions are tightly clustered, forking contexts may be exhausted. A performance comparison of different forking policies and confidence predictor designs for multipath can be found in [1].

**Multiple Path Contexts.** Multipath execution assumes multiple thread contexts, to allow several control flows to execute concurrently. In particular, the fetch unit must fetch from multiple paths each cycle. (Section 7.5 evaluates various multipath / fetch-bandwidth combinations.)

The essential hardware for each path is:

- a program counter
- a copy of, or its own port into, elements of the instruction-fetch unit, including the instruction cache, branch predictor, and confidence predictor
- a separate return-address stack—*not* a port into a unified one (Section 7.5.3)
- a register map, and shadow register maps sufficient for any unforked (conventionally speculated) branches which execution might have to unroll

The fetch unit tags instructions with a thread-ID. Since each thread has its own register map, renaming remains essentially unchanged from a conventional CPU; there are no renaming dependencies among threads. Predecode bits can be maintained in the instruction cache, indicating dependencies among instructions fetched for a particular thread. This further speeds up renaming and ensures that all the instructions fetched in a cycle can be renamed quickly.

The instruction window contains a mixture of instructions from active threads. Instructions are tagged with a thread identifier. One could instead provide separate instruction windows per path, but that approach would give poor performance when only a small number of threads are active, *i.e.*, when the program has few mispredicted branches.

Issue remains unchanged. Renaming ensures a coherent view of the register space, so instructions may arbitrate for execution as soon as their operands become ready. Context tags in the load-store queue ensure that threads see data flow through memory from only the appropriate path.

**Selective Misprediction Recovery.** In a single-path processor, the CPU handles a mis-speculated path by squashing all the instructions in the RUU after the mispredicted branch. In a multipath processor, however, instructions from many paths—including the correct path—may exist simultaneously and interleaved in the RUU. The CPU must have the ability to selectively squash only those instructions that belong to a particular mis-speculated path and its children. To accomplish this, every path has a unique *path ID* which encodes the forking history of recent branches using binary-prefix notation. New IDs are created when a branch forks, and IDs can be recycled. If a forking branch's ID is $x$, then the taken path's ID is $x1$, and the not-taken path's ID is $x0$. When that branch resolves, it broadcasts the correct path's ID, say $x1$. The CPU then squashes all instructions that follow the branch and are on the wrong path or a descendent, *e.g.* on paths $x0$, $x00$, $x01$; instructions from $x1$, $x10$, $x11$, etc. are spared. Non-forking, conventionally-speculated branches broadcast similarly, but for a mispredicted branch no instructions from the correct path exist. Squashed instructions turn themselves into NOPs. The resulting "holes" propagate like normal instructions but are ignored until commit, when they are reclaimed. A similar idea was independently proposed by Klauser, Paithankar, and Grunwald [51], and by Kol and Ginosaur [52].

The path IDs are implemented as circular bitmaps, with a global head pointer and a per-instruction tail pointer. The global head pointer indicates the oldest active forked branch. An instruction's tail pointer indicates how many subsequent branches have forked. Together, the pointers indicate what portion of any bitmap contains useful information: newer instructions, for example, have a tail pointer farther from the global head pointer, because more intervening branches have been seen. Since branches may resolve out of order, and the global head pointer cannot advance until the oldest forked branch retires, bitmaps could grow until the tail pointer would overtake the head. Forking halts at this point. But the bitmap length is equal to the depth of the forking tree: unless the tree is long and narrow and some branch takes an unusually long time to retire, bitmap length is not a problem. A bitmap needs to be at least as long as the $log_2$ of the maximum number of outstanding paths.

A per-branch pointer must be stored as well, indicating which bitmap position corresponds to each branch. When a branch retires and its thread's local head pointer matches the global head pointer, the global head pointer can be advanced.

### 7.2.2 Similarities to Conventional CPUs

It is important to note that the back end of a multipath CPU is essentially a conventional CPU augmented with extra issue and execution capacity to handle the extra running paths. For the bulk of this chapter, experiments use a "wide-open" back-end configuration chosen such that neither issue capacity nor functional units are a bottleneck. Then Section 7.6 considers cases with more limited back-end capacity.

The hardware, as described thus far, is similar to that used for simultaneous multithreading [99]. Although SMT is a complementary strategy that could be used in conjunction with multipath execution, that is beyond this chapter's scope. This work focuses solely on the performance benefits of multipath execution at branches, and corresponding fetch-bandwidth and confidence prediction requirements, rather than other aspects of program or workload parallelism. Wallace, Calder, and Tullsen examine a combined SMT/multipath organization in [107].

## 7.3 Simulation Technique

### 7.3.1 Simulator and Baseline Processor

These ideas have been evaluated using *HydraScalar*, which can also model multipath execution at the cycle-level. The baseline CPU modeled in these experiments resembles in most respects the one described in Chapter 2. The minimum misprediction latency is now eight cycles; one extra cycle is charged compared to the previous chapters, since multipath is such an aggressive technique. For direct branches, if only the branch target has been mispredicted, a dedicated adder in the decode stage calculates the correct target, and these misfetches suffer only a three-cycle penalty.

In the baseline configuration, 16 instructions can be fetched on each cycle, divided among the active threads. A particular thread can predict multiple branches per cycle and fetch past not-taken branches, but cannot fetch past taken branches.

The architectural registers (32 each for integer and floating-point) are separate from the rename registers, and are updated on commit. Renaming looks in a register mapping table to determine whether operands reside in the RUU or have been committed to architectural state. This mapping table is copied when a new path is forked, and a shadow copy of the table must be saved each time the processor speculates past a branch. If all a thread's shadow maps are full, fetch stalls for that thread. The baseline configuration allows 20 in-flight branches per thread. Each thread also has a private copy of the return-address stack, copied from the parent at fork time. The need for this is discussed in Section 7.5.

The experiments use three types of confidence prediction: omniscient, naive, and a hardware-based, ones-counting confidence predictor. The ones-counter is a 2K-entry table indexed by branch PC only; unlike the design described in [38], which combines the PC bits with branch-history bits. This gshare-style mechanism works well in conjunction with a gshare predictor but not with a hybrid predictor [61]. Each entry tracks the correctness of the last eight outcomes for branches indexing that entry; if two or more of the last eight outcomes were mispredicted, the confidence predictor returns a low-confidence output. The confidence predictor responds in a single cycle. This is plausible because this matches the response time of the branch predictor, and the confidence predictors modeled here use simpler hardware. Instruction fetch down the forked thread's speculative path can commence on the cycle after the fork decision.

The impact of multipath is dependent on the underlying branch predictor. This work models the now-familiar hybrid branch predictor configuration with non-speculative update that is described in Chapter 3 on page 44.

In order to focus on multipath behavior, most of the simulations reported here model an infinite processor from the decode stage on. This prevents instruction window size, decode / rename / issue bandwidth, and the like from obscuring the potential impact of fetch bandwidth or confidence prediction. Section 7.6 shows that an implementable configuration should also provide substantial speedups.

| | Warmup Insts | Branches per Instruction | | Branch Accuracies | |
|---|---|---|---|---|---|
| | | All | Cond | All | Cond |
| go | 926 M | 0.144 | 0.111 | 0.795 | 0.750 |
| gcc (cc1) | 221 M | 0.194 | 0.144 | 0.855 | 0.857 |
| compress | 2576 M | 0.202 | 0.133 | 0.921 | 0.880 |
| li (xlisp) | 271 M | 0.236 | 0.137 | 0.936 | 0.916 |
| perl | 601 M | 0.193 | 0.129 | 0.918 | 0.933 |

Table 7.1: Benchmark summary.
Statistics are from the post-warmup, 50 M-instruction simulations. Accuracy for "All" gives target-address accuracy for all branches, including indirect jumps; accuracy for "Cond" gives direction-prediction accuracy for conditional branches.

### 7.3.2 Benchmarks

Multipath execution can only benefit applications that suffer from poor branch prediction. Because this work focuses on fetch-bandwidth and confidence prediction requirements for multipath, the evaluations focus on the five SPECint95 benchmarks summarized in Table 7.1. The branch prediction accuracies in this table differ slightly from those in Chapter 2 due to the more aggressive fetch engine. Section 7.6 adds results from the remaining SPECint programs and shows that the suggested techniques benefit them as well. As always, data have been collected for a representative 50 million instruction simulation window from executions using the usual reference inputs.

## 7.4 Quantifying Opportunities for Multi-Path Execution

First, let us consider the performance benefits that multipath execution, at least in an idealized form, can offer. Figure 7.1 illustrates behavior under four different execution scenarios. In each graph, the bottom and top horizontal lines give baseline

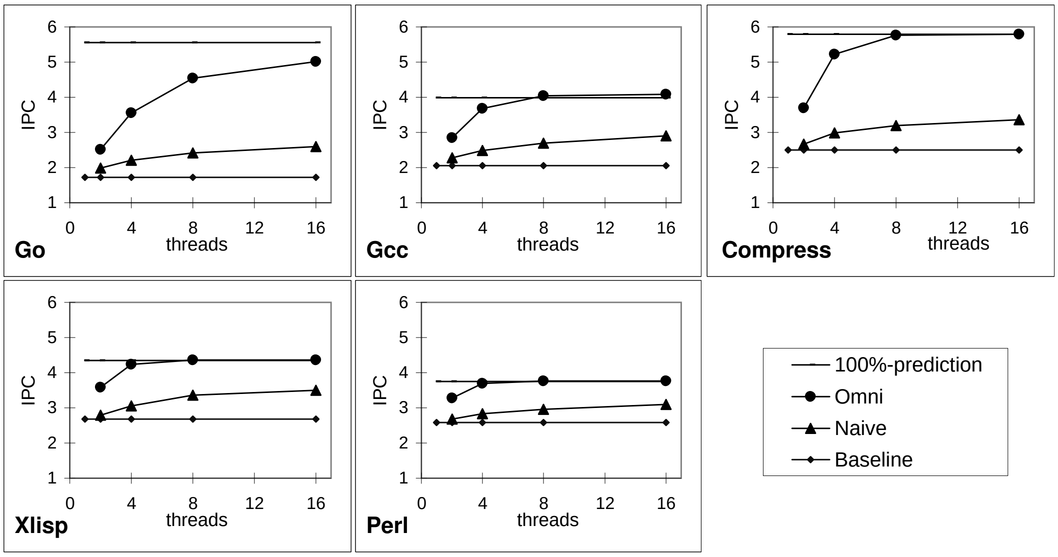


Figure 7.1: IPCs with omniscient and naive forking.
Each path receives as much fetch bandwidth as the single-path baseline processor (16 instructions/cycle). Single-path IPCs for baseline and 100%-direction-prediction configurations are shown as horizontal lines.

and idealized single-path performance. The idealized performance is the IPC that would result from having the directions (but not necessarily the target addresses) of conditional branches predicted with 100% accuracy.

The "naive" line represents forking every time a branch is encountered and a context is available. Even when each path is allowed to fetch as many instructions as the single-path baseline processor—*i.e.*, forking does not divide the fetch bandwidth—IPC rises slowly, falling well short of the ideal even with a large number of paths. If paths must instead share bandwidth, performance peaks at four paths (eight paths for *go*) and then declines.

The "omni" line represents omniscient forking. That is, the processor knows exactly when the branch predictor will mispredict, and forks precisely then so long as a context is available. Each path is again allowed to fetch as many instructions as the single-path baseline processor. Performance with this configuration reaches the

100% line with only a small number of paths. *Go* is an exception: it has so many mispredictions that it must fork more often in order to approach the ideal.

These plots show that forking selectively should improve substantially over naive forking. They also show that even with omniscient forking, a non-trivial number of paths are needed to eliminate branch mispredictions. Finally, the fact that performance does reach the 100% line using a moderate number of paths was a useful confirmation of HydraScalar's accuracy.

## 7.5 Hardware Effects

This section examines some front-end hardware issues that have strong leverage on multipath performance.

### 7.5.1 Fetch Bandwidth

Figures 7.2 and 7.3 show normalized IPCs for the five benchmarks with either omniscient or naive forking being used. For a particular benchmark, each layered group of bars represents a certain instruction-fetch bandwidth. Bandwidths of 8, 16 and 32 instructions per cycle are evaluated. No particular path receives priority; available bandwidth is split evenly among the paths. Within a layered group of bars, different shades represent different degrees of multipath. Notice that eight paths are not simulated when bandwidth is only eight instructions per cycle; individual paths receive so little fetch bandwidth that this policy is uniformly terrible. These data normalize all results to the 16-instructions/single-path configuration, which is why the other single-path speedups vary slightly from 1.0.

Omniscient-forking speedups depend heavily on fetch bandwidth for four- and eight-path configurations. These configurations see significant IPC speedups up to

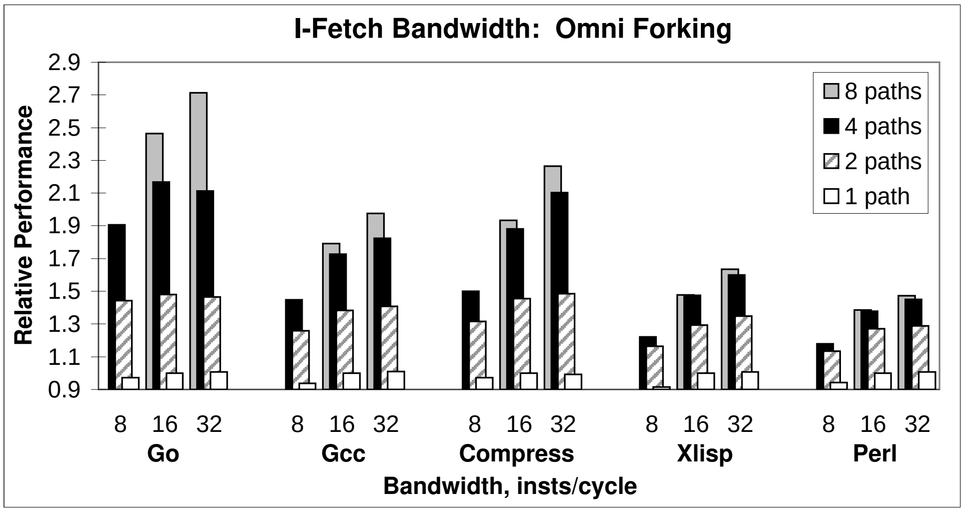


Figure 7.2: Application IPC as a function of instruction-fetch bandwidths using the omniscient forking policy. (Note non-zero y-axis origin, and that all results are normalized to one path/16-insts.)

32 instructions fetched per cycle. In contrast, the performance of the one- and two-path configurations levels off; fetching more than 16 instructions per cycle offers little performance benefit. While it is intuitive that exploring additional paths requires fetching more instructions, the graphs illustrate that the effect is not multiplicative with the number of paths.

Figure 7.3 now presents the same data for the naive forking policy. (Note the difference in y-axis scale.) This graph differs in two main ways. First, the IPCs using naive forking are dramatically lower across the board. Second, IPC does not monotonically increase with the number of paths explored if instruction fetch bandwidth is held constant. This is because naive forking allows path resources and fetch bandwidth to be allocated, in part, to paths that turn out to be useless. In the omniscient case, such wasted resources would not occur; forking only occurs when it helps performance. Like omniscient forking, benefits with naive forking depend on instruction-fetch bandwidth.

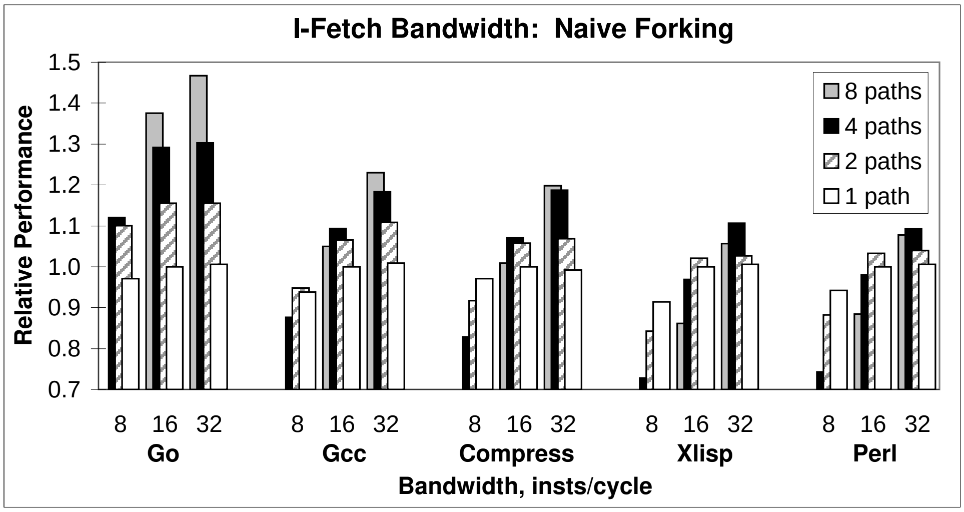


Figure 7.3: Application IPC as a function of instruction-fetch bandwidths using the naive forking policy.

### 7.5.2 Fetch-Priority Policy

To avoid wasting fetch resources along incorrect paths, heuristics can be explored for priority-based allocation of fetch bandwidth among the executing paths. While we cannot know the correct path *a priori*, we can note that the predicted path—the path indicated by the branch predictor—is correct most of the time. Since the paths other than the predicted one are often wrong, performance often improves when the predicted path gets somewhat more I-fetch bandwidth than other paths. Applied too extremely, paths starve and behavior reverts to the single-path case.

Three schemes are compared: a simple round-robin policy ("simple"), a round-robin policy that ensures the predicted path can fetch every cycle ("pred-pri"), and a predicted-path-priority policy that allows non-predicted paths to fetch only one cache line per cycle, while the predicted path gets any left-over bandwidth ("pred-extra"). Fetch schemes favoring the predicted path require that the predicted path can be known and looked up at any time. This can be implemented using a set of per-path bitmasks similar to path IDs.

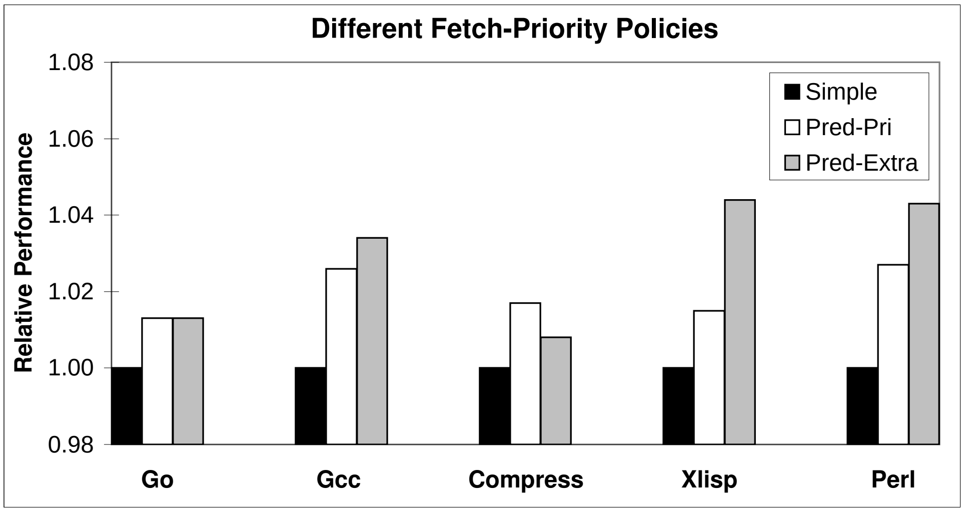


Figure 7.4: Speedups resulting from priority-based fetch policies: four paths/16 insts.

Figure 7.4 shows that giving more priority to the predicted path helps performance slightly— a few percent in most cases. Pred-extra outperforms the other two policies for all but one application. For these five benchmarks, *go* sees the least improvement from priority-based fetch schemes. This is because *go* has the least accurate branch prediction, so it is less likely to properly allocate fetch resources to the appropriate path.

### 7.5.3 Return-Address Stack

Return-address-stack accuracy can be an especially strong lever on multipath performance. A single, unified stack does not function properly in a multipath processor. With concurrent paths simultaneously modifying the stack, entries are popped and pushed by both correct and incorrect paths, making corruption almost certain. Not even full-stack checkpointing can prevent this. Figure 7.5 shows how this corruption might happen. For example, after a fork, both paths might encounter the same return instruction. One of these is legitimate, while the other occurs on a mis-speculated path, but both pop the stack. The second pop results in a return to the wrong subroutine level, and the stack has now been corrupted. In fact, the results presented in this section suggest that with multipath execution, corruption is severe enough that simply omitting the return-address stack and using the BTB leaves performance almost unchanged for most programs.

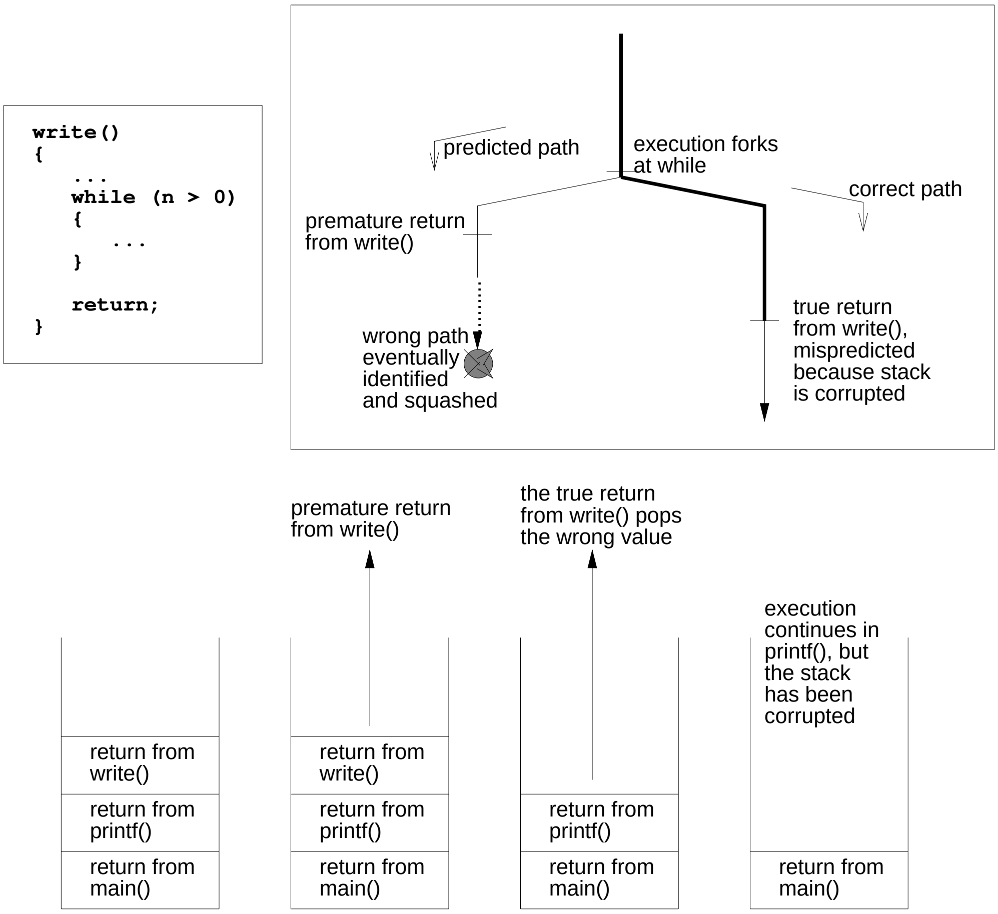


Figure 7.5: An example of how multipath execution can corrupt a unified return-address stack.

The best solution found simply gives each path a private copy of the return-address stack. Multipath execution already requires path contexts to record various state about each path: PC, shadow register maps, etc. The return-address stack merely adds an additional element in the path context. Note that multithreading also requires per-thread copies of the return-address-stack [35]. If simultaneous multithreading is used to provide multipath capability [107], the return-address stack copies will be used by both multithreading and multipath computation.

Copying the stack need not take place in a single cycle. If the new stack only receives a correct TOS pointer in the first cycle, it can already receive pushes from the new path. Pops, although returning an invalid address, will move the pointer correctly so that as stack entries are gradually copied, later pops can return correct values. Copying should of course start with the top of the stack.

An alternative approach uses a unified stack, but only allows one path to access the stack. Any other active paths must use the BTB. A variety of heuristics are possible for choosing the privileged path. This work uses the predicted path—the path that would have been followed if no forking occurred—as the one allowed to access the stack. This is still the path most likely to be correct, but the drawback to this approach is that during a misprediction, the wrong path has the privilege; after the misprediction resolves, the newly-recognized predicted path may find a corrupted stack.

Tables 7.2 and 7.3 compare the performance of four configurations: *(i)* no stack, *(ii)* a unified stack, *(iii)* a unified stack that only the predicted path can access, and *(iv)* per-path stacks. Table 7.2 presents return-prediction accuracies (note that for the no-stack case, this accuracy presents the return-address hit-rate in the BTB), and Table 7.3 presents performance in terms of speedup. Performance is normalized to the conventional, unified-stack case. Preceding experiments used a processor model with a wide-open back end, but these experiments use an aggressive but much more realistic design. Both two- and four-path processors are tested. The two-path processor can fetch, decode, issue, and commit 8 instructions each cycle; the four-path processor can fetch, decode, issue, and commit 16 instructions per cycle. The capacities of the

| | No Stack (use BTB) | | Unified Stack | | Unified Stack pred-access only | | Per-Path Stacks | |
|---|---|---|---|---|---|---|---|---|
| | 2 path | 4 path | 2 path | 4 path | 2 path | 4 path | 2 path | 4 path |
| go | 0.380 | 0.379 | 0.478 | 0.326 | 0.588 | 0.428 | 0.979 | 0.984 |
| m88ksim | 0.815 | 0.821 | 0.701 | 0.566 | 0.856 | 0.769 | 0.986 | 0.984 |
| gcc | 0.359 | 0.358 | 0.549 | 0.472 | 0.678 | 0.535 | 0.973 | 0.968 |
| compress | 0.511 | 0.533 | 0.884 | 0.788 | 0.972 | 0.901 | 1.000 | 1.000 |
| xlisp | 0.681 | 0.683 | 0.604 | 0.566 | 0.791 | 0.698 | 0.976 | 0.986 |
| ijpeg | 0.793 | 0.793 | 0.789 | 0.771 | 0.856 | 0.846 | 0.982 | 0.981 |
| perl | 0.579 | 0.579 | 0.525 | 0.495 | 0.806 | 0.712 | 0.958 | 0.950 |
| vortex | 0.475 | 0.475 | 0.827 | 0.807 | 0.895 | 0.898 | 0.998 | 0.998 |

Table 7.2: Return prediction accuracies for different stack organizations under multipath execution. Accuracies are for the correct path (committed return instructions) only.

| | No Stack (use BTB) | | Unified Stack | | Unified Stack pred-access only | | Per-Path Stacks | |
|---|---|---|---|---|---|---|---|---|
| | 2 path | 4 path | 2 path | 4 path | 2 path | 4 path | 2 path | 4 path |
| go | 1.003 | 1.022 | 1.000 | 1.000 | 1.013 | 1.013 | 1.057 | 1.069 |
| m88ksim | 1.051 | 1.124 | 1.000 | 1.000 | 1.068 | 1.093 | 1.131 | 1.214 |
| gcc | 0.959 | 0.973 | 1.000 | 1.000 | 1.030 | 1.015 | 1.110 | 1.142 |
| compress | 0.858 | 0.887 | 1.000 | 1.000 | 1.027 | 1.029 | 1.038 | 1.080 |
| xlisp | 1.039 | 1.054 | 1.000 | 1.000 | 1.096 | 1.062 | 1.205 | 1.263 |
| ijpeg | 1.000 | 1.001 | 1.000 | 1.000 | 1.001 | 1.002 | 1.004 | 1.005 |
| perl | 1.028 | 1.042 | 1.000 | 1.000 | 1.131 | 1.102 | 1.214 | 1.242 |
| vortex | 0.752 | 0.730 | 1.000 | 1.000 | 1.054 | 1.098 | 1.164 | 1.237 |

Table 7.3: Relative performance for different stack organizations under multipath execution. Two-path results are normalized to the two-path, unified-stack case, and four-path results to the four-path, unified-stack case.

RUU and load-store queue are 256 and 128 instructions, respectively. The level-one instruction and data caches are 256 KB and two-way associative, with 32-byte lines and two-cycle hit times. The unified, 16 MB, level-two cache has four-way associativity, 32-byte lines, and a 15-cycle hit time. The per-path return address stacks have 32 entries each. In all configurations, the stacks implement the misprediction-fixup mechanism proposed in the last chapter, namely restoring both the top-of-stack pointer and the top-of-stack entry's contents. Finally, note that although it is not necessary, the per-path stacks are always copied in a single cycle.

A unified stack achieves poor return hit rates, despite the use of top-of-stack pointer and top-of-stack contents fixup. For half the benchmarks, no stack is better than a unified stack, substantially so for *m88ksim* and *xlisp*. *Compress* and *vortex*, on the other hand, suffer badly if returns are only predicted from the BTB. None of these choices has any impact on *ijpeg*, so that program is not discussed further.

Using per-path stacks provides substantial gains for all the benchmarks. Hit rates return to near 100% levels (always reaching 100% with full-stack checkpointing, whose numbers are not presented here), and speedups range from 7% with *go* to 26% with *xlisp*. *Go's* speedup is low compared to its large improvement in hit rate, but *go* has fewer returns than most of the other programs, and time spent on return-address misses is in any case dwarfed by its high number of conditional-branch mispredictions.

A unified stack accessed only by the predicted path is a clear improvement over a simple unified stack, but is substantially inferior to per-path stacks. With two paths, the improvement is on average half that seen with per-path stacks. With four paths, hit rates decline compared to the two-path case, because the predicted path is less likely to be the correct one; speedups are sometimes higher for four paths than for two paths, because hit rates in the conventional stack decline too. Return mispredictions also tend to matter more in the four-path case, because four-way multipath is more effective at eliminating conditional-branch mispredictions. Another disadvantage to this approach is that, as further research leads to more selective forking, predicted-path schemes will become less effective.

## 7.6 Finite Processor Model

Except for the brief discussion of return-address stacks for multipath, the processor modeled up to this point has had a "wide-open" back end (unlimited functional units, RUU capacity, etc.) in order to isolate the fetch-bandwidth and confidence prediction requirements of multipath execution. In this section, the processor is limited to a hardware model that may be realizable in a technology generation or two.

As with the return-address stack, this finite processor can fetch, decode, issue, and commit 16 instructions each cycle. The capacities of the RUU and load-store queue are 256 and 128 instructions, respectively. The level-one instruction and data caches are 256 KB and two-way associative, with 32-byte lines and two-cycle hit times. The unified, 16 MB, level-two cache has four-way associativity, 32-byte lines, and a 15-cycle hit time. The per-path return address stacks have 32 entries each.

Figure 7.6 summarizes the results for a four-path run using this platform. It plots performance relative to a single-path baseline run using the limited configuration (this makes the "wide-open" speedups seem larger than those reported elsewhere in this chapter).

For all the benchmarks, limiting the back end has substantial effect, reducing the performance by 12.4%. Nevertheless, across the five benchmarks that have been the focus of this chapter, speedups from a realistic multipath configuration range from 8% to 31%, with an average of 16.9%.

Although *m88ksim, ijpeg*, and *vortex* have not been among the five focus appli-

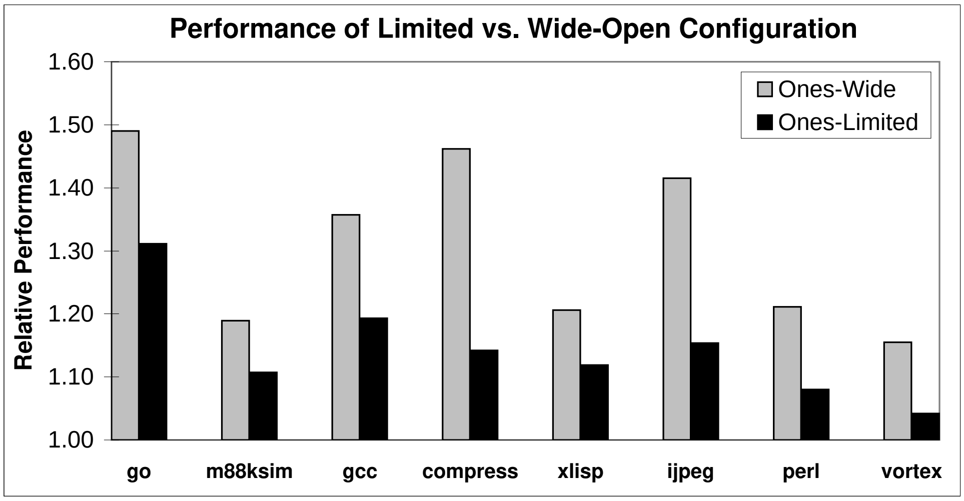


Figure 7.6: Relative performance of multipath execution with a wide-open and with a limited back-end configuration.

cations, these results include them to show that multipath execution benefits these programs as well. *M88ksim* achieves a branch prediction accuracy of 94.2%, *ijpeg* achieves 87.4%[1], and *vortex* achieves 97.8%. Speedups for these programs range from 4% for *vortex* to 12% for *ijpeg*, making the average speedup over the entire SPECint suite 14.4%. This suggests that a ones counter can control forking and not harm performance for programs with good branch prediction accuracy.

## 7.7 Conclusions and Future Work

As CPU issue widths become larger and pipelines become deeper, the performance costs of conditional branches become increasingly significant. Despite use of a sophisticated branch predictor, branch prediction accuracy is still so low that IPC would improve in the five focus applications by factors of 1.5 to 3.25 if the direction of conditional branches could be predicted with 100% accuracy. For these reasons, this

[1]But *ijpeg* has large basic blocks, resulting in a branch frequency about 1/3 that of the other programs (0.063 branches/inst).

work has proposed multipath-execution schemes and evaluated their performance on a range of programs.

While the hardware requirements of a four-path approach are not insignificant, this technique substantially complements other likely directions for future microprocessors. These include clustered approaches, multithreaded processors [56], and particularly simultaneous multithreading (SMT) [99]. Multipath speculation at conditional branches is largely an orthogonal way to make use of this hardware at times when other multithreading mechanisms do not. For example, Wallace, Calder, and Tullsen describe a system that combines SMT with multipath execution in [107].

This work suggests the following three conclusions: First, exploring performance trends for up to eight paths has shown that multipath execution can in the limit approach the performance of an idealized scheme in which conditional branches are predicted with 100% direction accuracy. But even with omniscient forking, this requires a non-trivial number of paths.

Second, at practical path counts (four or less), multipath execution offers speedups over single-path execution even with naive forking schemes. Using a simple but more selective ones-counting confidence predictor, rather than the naive scheme, affords speedups of up to 4–31% over the single-path case, with an average of 14.4% for the SPECint suite.

Finally, instruction fetch bandwidth and return-address stack configuration are two of the most important levers on performance. In addition, systems implementing priority-based fetch schemes can further improve performance by a few percent.

The significant performance benefits of multipath execution warrant further study, particularly in conjunction with multithreading and other forms of parallelism. Here three interesting avenues for future work are mentioned. First, better confidence prediction and better profiling will help identify forking opportunities more precisely, improving performance and reducing hardware requirements. Second, hardware demands could be reduced by hedging a forking decision in cases where the confidence prediction only weakly suggests a fork. For example, the fetch engine might prefetch some instructions from the alternate path without allocating a new path, or might allocate a new path and begin fetching, but give other paths much higher fetch priority. A third interesting avenue concerns interactions between predication [40, 75] and multipath. Predication can eliminate some hard-to-predict branches, especially those associated with small basic blocks and if-then-else structures that have no internal control-flow instructions. These techniques will reduce the forking capacity needed in a multipath processor.

# Chapter 8

# Conclusions

This dissertation has shown the degree to which branch prediction is a powerful lever over processor performance, and that the overall misprediction rate can be broken down according to the different sources of mispredictions. The dissertation then shows that *alloying* is an effective way to deal with wrong-history mispredictions, *speculative update with fixup* eliminates problems due to late update of the predictor's history, and that *multipath execution* is a promising technique for eliminating mispredictions from intrinsically hard-to-predict branches.

Branch mispredictions are a problem because each misprediction introduces a pipeline bubble of seven, ten, or more cycles, so even small misprediction rates hurt performance substantially. Furthermore, as Chapter 3 showed, mispredictions impede the ability of other processor structures to function effectively. The most striking example of this is the instruction window. With existing prediction accuracies, the instruction window cannot build up a large pool of instructions and cannot successfully expose instruction-level-parallelism (ILP). The ILP is present however [106], and better prediction can allow the instruction window to exploit that ILP.

Chapter 4 then measured the different sources of branch mispredictions according to a taxonomy of misprediction types. The data show that conflicts in the predictor's state tables and predictor training time are important contributors to the overall misprediction rate. These two sources of mispredictions have been well studied and many good solutions have been proposed [16, 20, 45, 57, 66, 82, 94]. But even if these two sources of mispredictions could be entirely eliminated, about half the overall misprediction rate remains untouched. The data show other, equally important contributors that have received much less attention in the research literature. Mispredictions due to wrong-history, improper update-timing, and intrinsically hard-to-predict branches must also be attacked in order to most effectively reduce the misprediction rate.

In addition to identifying this taxonomy of misprediction types, this thesis describes techniques for addressing these other sources of mispredictions.

## 8.1 Thesis Summary

### 8.1.1 Taxonomy

Mispredictions happen for different reasons. Solutions that only target a subset of the mispredictions can only reduce the overall misprediction rate to the extent that those eliminated mispredictions contribute to the overall rate. This is just a corollary of Amdahl's Law.

A taxonomy of mispredictions is therefore an important tool in devising better branch predictors. A taxonomy has at least two benefits. The first is that a taxonomy identifies all the sources of mispredictions and their importance. Efforts to improve prediction therefore know all the important problems that must be addressed by a successful solution. To the extent that the taxonomy comprises mutually exclusive categories, reductions in any one source of mispredictions are guaranteed to reduce the overall misprediction rate. Unfortunately, it is difficult to design a single predictor mechanism that can address all important sources of mispredictions. So the second benefit of a taxonomy is that it allows solutions to be tailored to each source of mispredictions. Ideally, the different mechanisms that result can cooperate or can even be composed into a unified mechanism.

This thesis identifies six mutually-exclusive sources of mispredictions:

1. *Conflict*: mispredictions due to destructive aliasing in one of the predictors state tables, *e.g.*, the BHT or PHT.

2. *Training-time*: mispredictions that occur because the predictor has not yet finished learning the branches' behavior.

3. *Wrong history*: mispredictions that occur because a branch needs either local or global history to predict successfully, but the available state does not provide that type of history (*e.g.*, a branch that needs global history when the predictor only tracks local history).

4. *Combined history*: similar to the previous category, but mispredictions that occur because a branch needs both types of history to be available simultaneously.

5. *Improper update timing*: unlike the previous mispredictions, which depend on *what* information the predictor contains, prediction accuracy also depends on *when* the predictor makes that information available. In particular, if the predictor's history is updated late, or if it is updated speculatively without a mechanism to correct errors from mispredictions, the history becomes inaccurate and causes mispredictions.

6. *Remaining mispredictions*: some mispredictions are intrinsically difficult to remove, either because the corresponding branches behave randomly, or because

they exhibit behavior that reasonable predictors cannot capture (like very long repeating patterns). Some technique that does not use prediction mechanisms is the easiest way to remove these mispredictions.

The importance of conflict and training-time mispredictions is known, and, as mentioned, a number of predictor mechanisms have been proposed that specifically target these problems. But Chapter 4 shows that these two problems typically account for less than half the overall misprediction rate. Wrong-history and "remaining" mispredictions are often as important, if not more important. Chapters 5 and 6 show that improper update timing is a further important source of mispredictions. The techniques proposed in this thesis have been tailored to address these other, still-important misprediction types.

### 8.1.2 Tailored Branch-Prediction Solutions

**Alloying.** After identifying the importance of wrong-history mispredictions, Chapter 4 shows how a simple change to existing two-level predictors—*alloying* global and local history bits in the same PHT index—can reduce wrong-history mispredictions. Hybrid prediction also reduces wrong-history mispredictions, but has several drawbacks. It cannot attack the minor category of combined-history mispredictions. It suffers from mispredictions if the hybrid predictor's selector does not behave well. And hybrid prediction performs badly for small hardware budgets. Alloying provides robust performance across all the benchmarks shown, and does so for both very large and very small hardware budgets. Regardless of predictor size, alloying outperforms competing two-level organizations. For a large hardware budget, alloying provides competitive performance compared to hybrid prediction. For a small hardware budget, alloying substantially outperforms hybrid prediction.

**Speculative update with fixup.** Chapter 5 shows that the performance of a processor using a global history predictor suffers by 10% on average for the SPECint95 suite if that predictor implements non-speculative (late) update. Naive speculative (early) update is much worse, because mispredictions add incorrect state to the predictor and degrade its accuracy much more than does late update. But speculative update with *fixup* provides the appearance of omniscient, early update, eliminating the 10% penalty that late update causes. The required fixup merely involves saving and restoring the global history register's contents for each branch. A similar mechanism benefits local history predictors, and requires saving and restoring any modified BHT entries in a future file. This provides a 5% performance improvement compared to late update.

Return instructions, which unlike branches are unconditional control-flow transfers, also benefit from speculative update with fixup. For most programs, a return-address stack can predict return targets with 100% accuracy, but only with an omniscient early-update mechanism. Prior work has described some ways to approximate that using speculative update with fixup [44, 65], and Chapter 6 describes a new, very simple heuristic that saves and restores just the top-of-stack information for the return-address stack. This mechanism is an approximation which recognizes that deeper stack entries are rarely corrupted and thus do not need to be protected. As a result, this simple technique provides near-100% accuracies. The corresponding performance improvement averages 5.5% for SPECint95, a speedup that is additive with those obtained by applying speculative update with fixup to the conditional-branch history.

**Multipath execution.** Multipath execution targets hard-to-predict branches by *forking*: simultaneously executing both paths of these branches. This ensures that the correct path begins execution as soon as possible, and the incorrect path can be squashed once the branch resolves. Hardware requirements are proportional to the number of paths, and that depends on how carefully the forking decision is made. Aggressive forking wastes path resources, while conservative forking may fail to fork on a mispredicted branch. With a simple dynamic *confidence predictor* [1, 29, 38] that uses a structure similar to conventional branch predictors to estimate when mispredictions will occur, Chapter 7 observes speedups of 14.4% for the SPECint suite using a four-path processor with 16-way issue. This aggressive hardware is expensive, but dovetails well with simultaneous multithreading techniques [100, 107]. Furthermore, as other techniques for eliminating hard-to-predict branches—like predication—improve, and as forking decisions improve, the hardware requirements of multipath will shrink, and its efficacy will likely improve.

**Combining these solutions** Most of these techniques can be combined into a single structure, showing how the taxonomy can focus design efforts while still yielding compatible solutions. Proposed techniques for reducing PHT conflicts or training-time mispredictions [16, 20, 45, 57, 66, 82, 94] should work equally well with a single type of history or with alloyed history. Furthermore, speculative update with fixup is a general approach that can be integrated into most predictor designs, including all those discussed in this dissertation. This leaves only multipath execution, which is necessarily separate, requiring some mechanism like a confidence predictor to determine whether a conventional branch prediction should be made or whether multipath execution should be invoked.

Figures 8.1–8.4 demonstrate that the benefits of alloying and speculative history update are indeed cumulative. For the eight SPECint benchmarks plus the four other integer benchmarks used in Chapter 4, these figures compare the performance of a conventional GAs predictor with non-speculative (late) update, the same GAs predictor with speculative history update and fixup, and an MAs predictor that also uses speculative history update and fixup. GAs with speculative update provides fixup by saving and restoring the global history register with each branch prediction. MAs with speculative update retains the global-history fixup, and adds an OBQ to fixup local history.

Figures 8.1 and 8.2 present performance for a processor with 4 integer instructions/cycle of issue bandwidth, using the configuration described in Chapter 2. Figures 8.3 and 8.4 present performance for a next-generation, 8-wide processor, using the configuration described in Chapter 5. In both cases, the processor has enough shadow state to accommodate up to 32 in-flight branches. As usual, the misprediction penalty is seven cycles (and two cycles for simple misfetches). These experiments were conducted using HydraScalar, the cycle-level simulator, and so performance is measured in IPC. Results are also presented in terms of speedup, giving performance normalized to that of the GAs predictor using late update. Note that the IPCs for global history differ from those presented in Chapter 5 because the predictor studied there was a GAg predictor not a GAs predictor. (The GAs predictor adds branch-address bits to the PHT index to reduce PHT conflicts.)

| GAs | MAs | | |
|---|---|---|---|
| index | index | BHT | PHT |
| 8g, 7a | 9g, 4p, 3a | 8K entries | 16K entries |

Table 8.1: 64 Kbit GAs and MAs predictor configurations.
BHTs are direct-mapped. “g” indicates the number of global-history bits, “p” local-history bits, and “a” address bits.

These experiments were performed using the best 64 Kbit GAs and MAs predictor configurations found in Chapter 4. Those configurations are repeated here in Table 8.1. The BHTs used here are direct-mapped, however, instead of the more aggressive

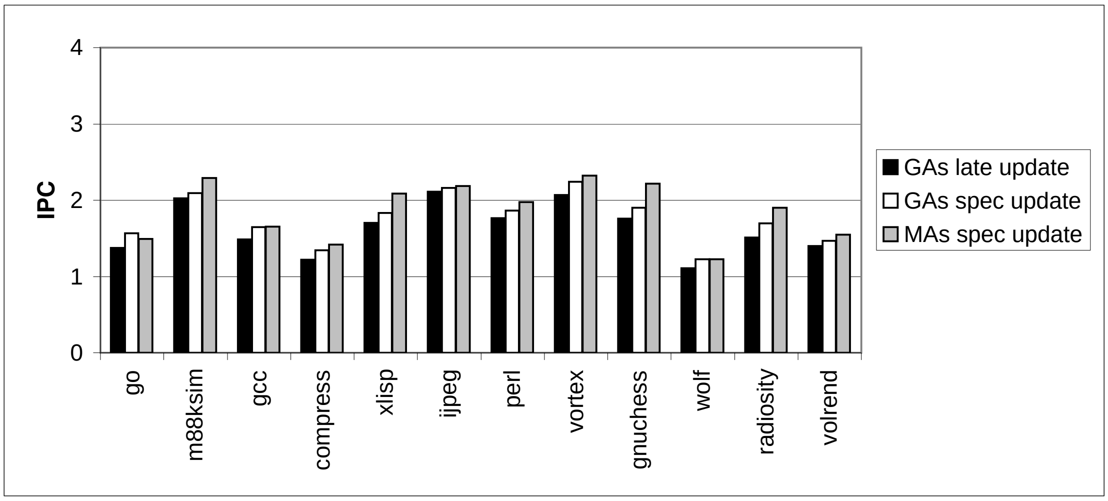


Figure 8.1: Performance (in IPC) of GAs with non-speculative (late) update, GAs with speculative update and fixup, and MAs with speculative update and fixup, for a 4-issue processor.

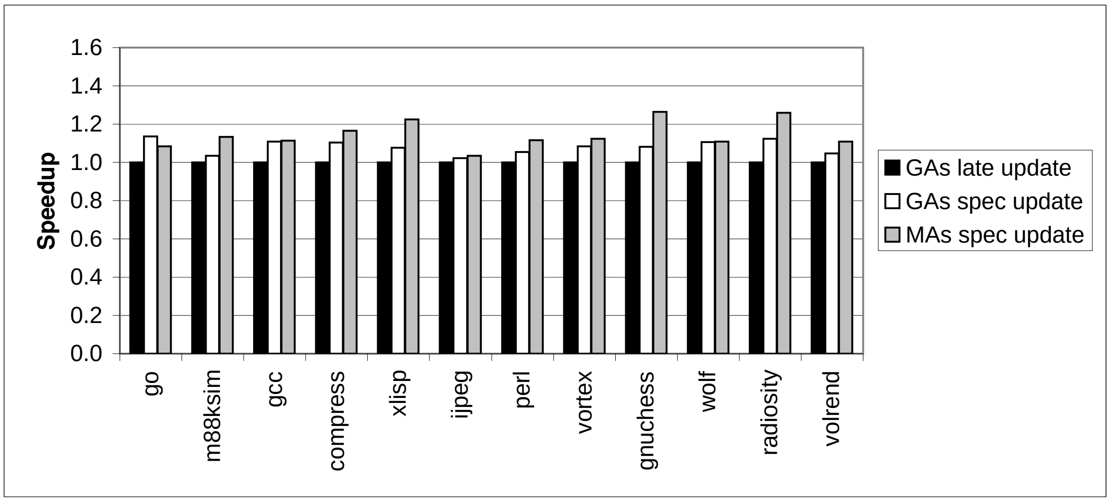


Figure 8.2: Relative performance (speedup) for a 4-wide processor, normalizing the above IPCs to the performance of the baseline, GAs with late update.

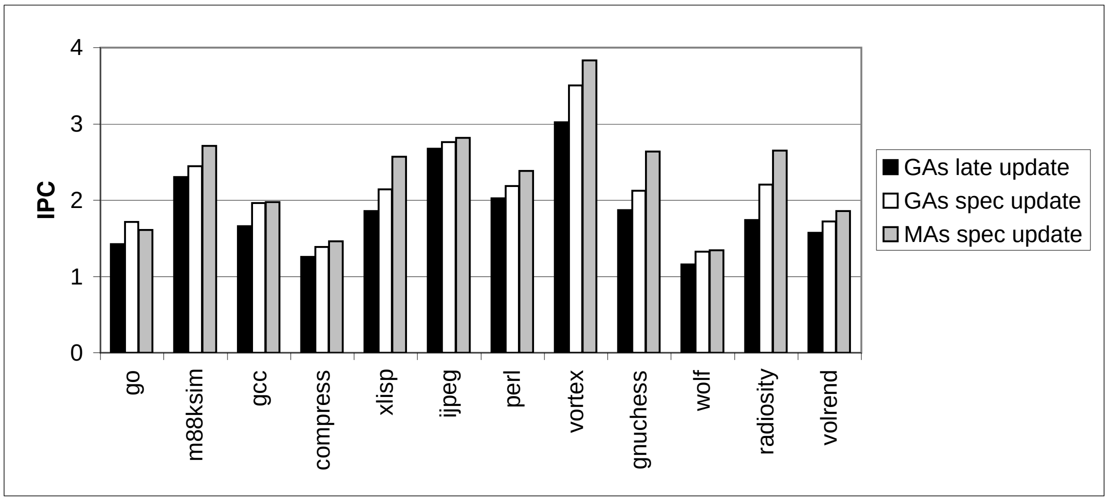


Figure 8.3: Performance (in IPC) of GAs with non-speculative update, GAs with speculative update and fixup, and MAs with speculative update and fixup, for an 8-issue processor.

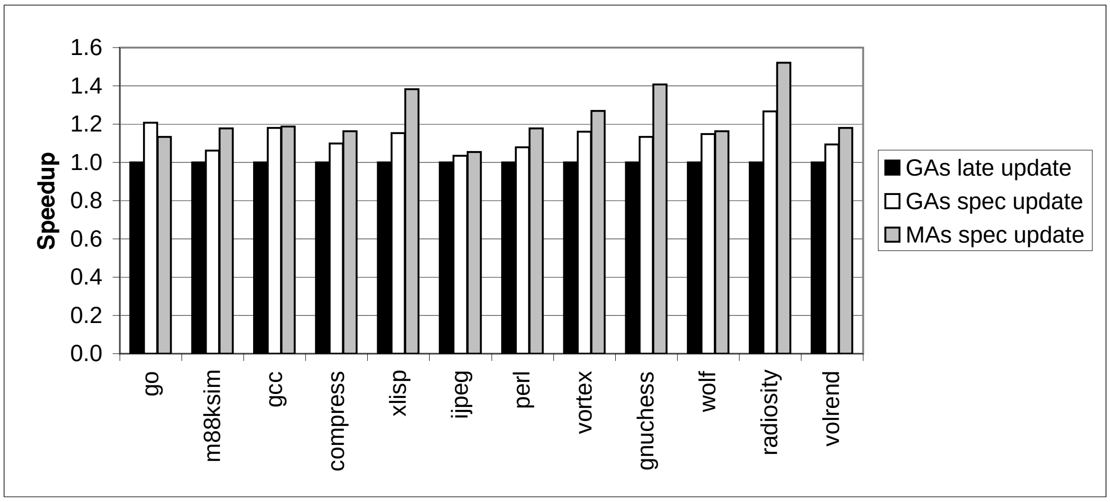


Figure 8.4: Relative performance (speedup) for an 8-wide processor, normalizing the above IPCs to the performance of the baseline, GAs with late update.

set-associative configuration from that chapter.

Speculative update and alloying not only combine easily, but also to great effect. This is true for both the 4-wide and 8-wide processors. Conflicts do cause *go* to perform worse with MAs than with GAs. Otherwise, as each feature is added—first the speculative update, then the alloying—performance improves substantially for every benchmark. Taking the benchmark set as a whole, MAs with spec-update is on average 8.7% better than GAs with speculative update for an 8-wide processor, and 23.5% better than GAs with late update.

## 8.2 Future Directions

The work presented here can be extended in many ways, four of which are described below.

### 8.2.1 Refining and Extending the Taxonomy

The most obvious avenue for future work is improving the taxonomy. Obvious additions to the taxonomy include mispredictions due to history length (the type of history is correct, but the history is too long or too short) and mispredictions due to noise in the history. Although obvious, it is not yet clear how to measure these categories. In addition to these, a unified measurement framework that includes timing effects would be another useful step forward; this thesis treated update timing separately because cycle-level simulations are expensive.

Another contribution would be to more closely understand which mispredictions are eliminated by each potential solution. Hybrid and alloyed prediction both reduce wrong-history mispredictions, but they may possibly eliminate different wrong-history mispredictions. Instantaneous characterizations of prediction streams like the one

Weikle *et al.* proposed for characterizing cache streams [109] might identify simple but localized conditions under which one solution is better than another. Such a technique might also suggest synergistic ways to combine alternative techniques.

Yet another interesting direction in which to extend the taxonomy is to include *positive* effects. For example, some aliasing is constructive, so when conflict-reduction techniques may not achieve as large a benefit as expected if they remove many constructive conflicts in addition to removing destructive conflicts. A characterization of effects like these would further improve understanding of branch predictor behavior.

### 8.2.2 Better Measurement and Simulation Techniques

HydraScalar, which is quite fast compared to other cycle-level simulators, still only simulates about 100 M instructions per hour, a slowdown of about 3600. Chapter 2 describes how to cope with this by obtaining a short but representative sample of execution, but this still does not include operating system or multi-tasking behavior. Currently SimOS is the only simulation system providing that capability [76]. Computer architects would benefit greatly from work on adding system and multi-tasking behavior to simulations without substantially extending simulation times; reducing overall simulation times; providing more efficient ways to verify the representativeness of chosen simulation windows; and on improving sampling methods to provide short, easy-to-obtain, but provably representative samples.

### 8.2.3 Better Use of Branch History

Global history is inefficient; many bits in the global history register are irrelevant for a particular branch's prediction. Not only does this increase training time (a prediction must be learned for each possible combination of these "don't-care" bits), but older yet more relevant history bits are displaced by newer don't-care bits [21]. In addition some bits—like those corresponding to branches that are always-taken or always-not-taken—never have predictive value. Better schemes for tracking branch history can pack more useful information into the same history width. This may also reduce harmful aliasing, because reducing the number of don't-care bits will reduce the footprint that individual branches occupy in the predictor's tables.

A second way to improve the efficacy of global branch history—and a useful behavior to incorporate into a taxonomy—is to minimize the different positions that a history bit corresponding to some particular branch can occupy in the history register. One problem with branch predictors that store only branch outcomes (i.e., taken/not-taken) in the history registers is that a particular branch can be reached by different paths through the program. This means that particular history bit positions have different meanings depending on the current path, and harmful aliasing among similar patterns can result. Encoding some branch-address bits with the outcome may help [69], but space requirements restrict the number of bits that can be used per history entry. An intriguing alternative is to use the compiler or a binary-rewriting tool to regularize path lengths. This eliminates aliasing from different path lengths by inserting dummy branches, so that regardless of the path followed, a particular branch always occupies the same position in the history. Combining path regularization with code placement techniques [8] should reduce aliasing even further.

### 8.2.4 Improving Multipath

Multipath remains an expensive solution, but one which dovetails closely with simultaneous multithreading [100, 107]. It attacks mispredictions that cannot be removed by any prediction-based technique or by compiler-based branch-removal techniques like predication. The most pressing need for improving multipath execution is better forking decisions, perhaps via better confidence estimation techniques [1, 29, 38]. Another avenue for improving multipath performance is to implement different *degrees* of multipath aggressiveness, so that fewer hardware resources are used for branches that need multipath less. For example, branches that warrant multipath execution but have higher confidence or that are not on critical code paths might only result in prefetching the non-predicted path, while the branches with the lowest confidence or that are on critical code paths receive aggressive multipath treatment, perhaps even receiving boosted priority compared to other paths. A final avenue for improving multipath execution involves compiler assistance. The compiler can help multipath by removing branches, by rearranging instructions on alternative branch paths to reduce hardware demands when that branch forks, and by splitting branches when that produces a branch with more consistent behavior.

## 8.3 In Summary

Further improvements in branch prediction are paramount to obtaining better processor performance. This thesis has shown the benefits of classifying the sources of branch mispredictions, so that all the important ones can be identified and attacked. Developing a taxonomy also leads to a better understanding of predictors' behavior and may directly suggest some useful solutions. While there is ample room to refine and extend the taxonomy presented here, it has already shown that future branch prediction work should not target conflict mispredictions—as much current research has done—but instead should focus on better ways to track and learn branch behavior, and on better techniques for eliminating mispredictions associated with inherently hard-to-predict branches.

# Bibliography

[1] P. S. Ahuja, K. Skadron, M. Martonosi, and D. W. Clark. Multi-path execution: Opportunities and limits. In *Proceedings of the 12th International Conference on Supercomputing*, pages 101–08, July 1998.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–89, Jan. 1983.

[3] D. I. August, D. A. Connors, J. C. Gyllenhaal, and W. W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 84–93, Feb. 1997.

[4] T. Austin. Personal communication, Mar. 1999.

[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Tech. Report TR-1308, University of Wisconsin-Madison Computer Sciences Dept., July 1996.

[6] M. G. Butler and Y. Patt. A comparative performance evaluation of various state maintenance mechanisms. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 70–79, Dec. 1993.

[7] B. Calder and D. Grunwald. Fast & accurate instruction fetch and branch prediction. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 2–11, May 1994.

[8] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–51, Oct. 1994.

[9] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Jan. 1994.

[10] P.-Y. Chang, M. Evers, and Y. N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. In *Proceedings of the 1996 International Conference on Parallel Architectures and Compilation Techniques*, pages 48–57, Oct. 1996.

[11] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 252–57, Dec. 1995.

[12] P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 274–83, June 1997.

[13] Digital Semiconductor. *DECchip 21064/21064A Alpha AXP Microprocessors: Hardware Reference Manual*, June 1994.

[14] Digital Semiconductor. *Alpha 21164 Microprocessor: Hardware Reference Manual*, Apr. 1995.

[15] K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 167–78, July 1998.

[16] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–77, Dec. 1998.

[17] J. Emer. Personal communication, June 1997.

[18] J. Emer. Personal communication, Mar. 1998.

[19] J. Emer and N. Gloy. A language for describing predictors and its application to automatic synthesis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 304–14, June 1997.

[20] 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*, pages 3–11, May 1996.

[21] 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*, pages 52–61, June 1998.

[22] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–43, May 1997.

[23] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[24] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Oct. 1992.

[25] A. Glew. Personal communication, Mar. 1998.

[26] N. Gloy, C. Young, J. B. Chen, and M. D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 12–21, May 1996.

[27] A. M. Gonzalez. A survey of branch techniques in pipelined processors. *Microprocessing & Microprogramming*, 36(5):243–57, Oct. 1993.

[28] D. Greenley et al. UltraSPARC$^{(TM)}$: The next generation superscalar 64-bit SPARC. In *Proceedings of CompCon '95*, pages 442–50, Mar. 1995.

[29] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 122–31, June 1998.

[30] D. Grunwald, D. Lindsay, and B. Zorn. Static methods in hybrid branch prediction. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 222–29, Oct. 1998.

[31] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, pages 9–15, Feb. 16, 1995.

[32] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, Oct. 28, 1996.

[33] E. Hao, P.-Y. Chang, and Y. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 228–32, Nov. 1994.

[34] T. H. Heil and J. E. Smith. Selective dual path execution. Tech. Report, University of Wisconsin-Madison Dept. of Electrical and Computer Engineering, Nov. 1996.

[35] S. Hily and A. Seznec. Branch prediction and simultaneous multithreading. In *Proceedings of the 1996 International Conference on Parallel Architectures and Compilation Techniques*, pages 169–73, Oct. 1996.

[36] W. Hwu and P. Chang. Inlining function expansion for compiling C programs. In *Proceedings ACM SIGPLAN 1989 Conference on Programming Languages Design and Implementation*, pages 246–55, June 1989.

[37] V. S. Iyengar and L. H. Trevillyan. Evaluation and generation of reduced traces for benchmarks. Research Report RC 20610, IBM, Oct. 1996.

[38] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 142–52, Dec. 1996.

[39] Q. Jacobsen, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 14–23, Dec. 1997.

[40] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 100–13, Dec. 1996.

[41] T. L. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–26, June 1997.

[42] N. P. Jouppi and P. Ranganathan. The relative importance of memory latency, bandwidth, and branch limits to performance. In *The Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997. http://ayer.CS.Berkeley.EDU/isca97-workshop.

[43] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–82, Apr. 1989.

[44] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt. Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution. *International Journal of Parallel Programming*, 25(5):363–83, Oct. 1997.

[45] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 156–66, June 1998.

[46] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–41, May 1991.

[47] J. Kalamatianos and D. R. Kaeli. Predicting indirect branches via data compression. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 272–81, Dec. 1998.

[48] R. E. Kessler, M. D. Hill, and D. A. Wood. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. Tech. Report TR-1048, University of Wisconsin Computer Sciences Department, Sept. 1991.

[49] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the 1998 International Conference on Computer Design*, pages 90–95, Oct. 1998.

[50] S. Kim and G. Tyson. Analyzing the working set characteristics of branch execution. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 49–58, Dec. 1998.

[51] A. Klauser, V. Paithankar, and D. Grunwald. Selective eager execution on the PolyPath Architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 250–59, July 1998.

[52] R. Kol and R. Ginosaur. Kin: A high performance asynchronous processor architecture. In *Proceedings of the 12th International Conference on Supercomputing*, pages 433–440, July 1998.

[53] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, June 1981.

[54] S. Laha, J. H. Patel, and R. K. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Transactions on Computers*, 37(11):1325–1336, Nov. 1988.

[55] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[56] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Oct. 1994.

[57] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 4–13, Dec. 1997.

[58] S. Mahlke and B. Natarajan. Compiler synthesized dynamic branch prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 153–164, Dec. 1996.

[59] S. A. Mahlke et al. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 217–27, Dec. 1994.

[60] S. A. Mahlke, R. E. Hank, J. E. McCormick, et al. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 138–49, June 1995.

[61] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 132–41, June 1998.

[62] M. Martonosi, A. Gupta, and T. Anderson. Effectiveness of Trace Sampling for Performance Debugging Tools. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 248–59, May 1993.

[63] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–155, Oct. 1994.

[64] S. McFarling. Combining branch predictors. Tech. Note TN-36, DEC WRL, June 1993.

[65] S. McMahan, Cyrix Corp. Branch processing unit with a return stack including repair using pointers from different pipe stages. U.S. Patent No. 5,706,491, Jan., 1998.

[66] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 292–303, June 1997.

[67] MIPS Technologies. *MIPS R10000 Microprocessor User's Manual*, June 1995. Version 1.0.

[68] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference*

*on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[69] R. Nair. Dynamic path-based correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 15–23, Dec. 1995.

[70] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Oct. 1992.

[71] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 165–75, Dec. 1996.

[72] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 120–29, Apr. 1994.

[73] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[74] C. Price. *MIPS IV Instruction Set, Revision 3.1*. MIPS Technologies, Inc., Mountain View, CA, Jan. 1995.

[75] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *Computer*, 22(1):12–35, Jan. 1989.

[76] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995.

[77] 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 IEEE/ACM International Symposium on Microarchitecture*, pages 24–34, Dec. 1996.

[78] E. Rotenberg, Q. Jacobsen, Y. Sazeides, and J. E. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, Dec. 1997.

[79] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[80] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–73, June 1997.

[81] Y. Sazeides and J. E. Smith. Modeling program predictability. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 73–84, June 1998.

[82] S. Sechrest, C.-C. Lee, and T. Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 22–32, May 1995.

[83] S. Sechrest, C.-C. Lee, and T. Mudge. The role of adaptivity in two-level adaptive branch prediction. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 264–69, Dec. 1995.

[84] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*. To appear.

[85] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 259–71, Dec. 1998.

[86] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 144–155, Feb. 1997.

[87] K. Skadron, D. W. Clark, and M. Martonosi. Speculative updates of local and global branch history: A quantitative analysis. Tech. Report TR-589-98, Princeton University Dept. of Computer Science, Dec. 1998.

[88] K. Skadron, M. Martonosi, and D. W. Clark. Alloying global and local branch history: Taxonomy, performance, and analysis. Tech. Report TR-594-99, Princeton University Dept. of Computer Science, Jan. 1999.

[89] K. Skadron, M. Martonosi, and D. W. Clark. Selecting a single, representative sample for accurate simulation of SPECint benchmarks. Tech. Report TR-595-99, Princeton University Dept. of Computer Science, Jan. 1999.

[90] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–48, May 1981.

[91] M. D. Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University, Nov. 1992.

[92] M. D. Smith, M. Johnson, and M. Horowitz. Limits on Multiple Instruction Issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, Apr. 1989.

[93] G. S. Sohi and A. S. Vajapeyam. Instruction issue logic for high-performance, interruptible pipelined processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, June 1987.

[94] 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*, pages 284–91, June 1997.

[95] S. Srinivasan and A. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 148–59, Dec. 1998.

[96] A. R. Talcott, M. Nemirovsky, and R. C. Wood. The influence of branch prediction table interference on branch prediction scheme performance. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, pages 89–96, June 1995.

[97] A. R. Talcott, W. Yamamoto, M. J. Serrano, R. C. Wood, and M. Nemirovsky. The impact of unresolved branches on branch prediction scheme performance. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 12–21, Apr. 1994.

[98] The Standard Performance Evaluation Corporation. WWW Site. http://www.specbench.org, Dec. 1996.

[99] D. Tullsen, S. Eggers, J. Emer, et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[100] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[101] G. Tyson and M. Farrens. Evaluating the effects of predicated execution on branch prediction. *International Journal of Parallel Programming*, 24(2):159–86, 1996.

[102] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. Tech. Report CSE-TR-346-97, University of Michigan, Dept. of Electrical Engineering and Computer Science, 1991.

[103] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 345–56, June 1995.

[104] A. Uht and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 313–25, Dec. 1995.

[105] A. K. Uht, V. Sindagi, and S. Somanathan. Branch effect reduction techniques. *Computer*, 30(5):71–81, May 1997.

[106] D. W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–88, Apr. 1991.

[107] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 238–49, July 1998.

[108] C. F. Webb. Subroutine call/return stack. *IBM Technical Disclosure Bulletin*, 30(11), Apr. 1988.

[109] D. A. B. Weikle, S. A. McKee, and W. A. Wulf. Caches as filters: A new approach to cache analysis. In *Proccedings of the Sixth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 19–24, July 1998.

[110] K. M. Wilson and K. Olukotun. Designing high bandwidth on-chip caches. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 121–32, June 1997.

[111] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings*

*of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[112] D. A. Wood, M. D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 79–89, June 1991.

[113] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[114] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–34, May 1992.

[115] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–66, May 1993.

[116] 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*, pages 276–86, June 1995.

[117] C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–41, Oct. 1994.