Speculative Updates of Local and Global Branch History: A Quantitative Analysis

Kevin Skadron SKADRON@CS.VIRGINIA.EDU
Dept. of Computer Science
University of Virginia
Charlottesville, VA 22903

Margaret Martonosi MRM@EE.PRINCETON.EDU
Douglas W. Clark DOUG@CS.PRINCETON.EDU
Depts. of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ 08544

Abstract

In today's wide-issue processors, even small branch-misprediction rates introduce substantial performance penalties. Worse yet, inadequate branch prediction creates a bottleneck at the fetch stage, restricting other opportunities for improving performance. The choice of how to predict conditional-branch outcomes is the primary lever on prediction accuracy. But the choice of *when* to update the predictor with branch outcomes is a second powerful lever, and the subject of this paper. In history-based predictors like *gshare*, many mispredictions result from commit-time update of the history: typical pipelined processors predict branches in the fetch stage, but update the predictor in the commit stage, making the predictor's state temporarily out-of-date. As pipelines grow longer—in particular, when branches can spend many cycles in the instruction window waiting to issue—this problem becomes worse. Prior work on this subject has discussed the need for speculative update in a global-history-based predictor; this paper evaluates speculative history update for both global-history and local-history predictors of various configurations, using cycle-level simulation to show the effects of prediction accuracy and update time on overall performance. The results show the importance of speculative history update, but also that speculative update requires suitable fixup mechanisms for repairing state that has been corrupted by mispredictions. A number of such mechanisms are discussed.

1. Introduction

Achieving the highest possible branch prediction accuracies is critical to good processor performance. In wide-issue, deeply-pipelined processors, a single misprediction creates a pipeline bubble that can waste the opportunity to execute 15, 30, or more instructions. For this reason, research continues to explore new techniques, especially for *dynamically* predicting branch outcomes. Dynamic schemes do not require instruction-set modifications, and they learn run-time behavior that compilers cannot account for, in particular variations resulting from different input data.

Two-level-adaptive schemes [1] 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 2-bit counters described by Smith [2] and found in several recent processors [3, 4]). Two-level schemes now appear in many high-performance processors: the AMD K6 and Athlon [5, 6] and the soon-to-be-released UltraSPARC III [7] use *gshare* [8] predictors; the Pentium-Pro/Pentium-II [9] also uses a two-level scheme, but of a confidential nature; and the Alpha 21264 [10, 11] uses two different two-level predictors in conjunction with a selector that chooses between the components [8, 12].

Most branch-prediction studies use 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; the minimum misprediction latency in the Alpha 21264, for example, is 7 cycles. During this 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.

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 paper 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.

Related Work. 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 [1], state that speculative update improves prediction accuracy but do not describe mechanisms for repair after mispredictions. Talcott *et al.* [13] 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 [14] 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 (see Section 3.1). Their results consider a GAp[1] scheme, a theoretical predictor that maintains global history but provides each branch with its own table of 2-bit counters (the *pattern history table*, or PHT). Jourdan *et al.* [15] 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. Their discussion mostly envisions storing necessary repair state in the instruction reorder buffer, as it draws on mechanisms for recovering register state that are described in [16]. 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. On the other hand, the Jourdan paper further showed that speculative update and repair in the *return-address stack* have a substantial effect on performance. Skadron *et al.* [17] extended this work, describing a simple repair return-address-stack repair mechanism that simply saves and restores the top-of-stack pointer and the top-of-stack contents. It nearly eliminates return mispredictions, with corresponding speedups of 4.5–8.7% in the SPECint95 benchmarks [18] on a processor similar to the Alpha 21264.

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 [11]. 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.

Contributions. This paper presents, for both global- and local-history schemes, a systematic evaluation, using eleven SPEC95 benchmarks, of how much speculative history update can improve performance. The work explores these issues as a function of branch predictor size and as a function of the number of branches

[1] Global history, Adaptive PHT, Per-address PHT

allowed to be in flight at once. This detailed data substantially extends prior characterizations of speculative update's importance for global-history based predictors [13, 14, 15], and to our knowledge is the first such published data for local-history based predictors.

Our 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, although this effect plateaus for most integer benchmarks at a level of 4–8 in-flight branches.

This paper 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 [8].

The rest of this paper is organized as follows. The next section describes our simulation techniques and the benchmarks we choose. Section 3 discusses speculative-update effects in a global-history predictor, and Section 4 discusses them in a local-history predictor. Section 5 reports results for a hybrid predictor design, and Section 6 concludes the paper.

2. Simulation Methodology

2.1 Simulator

We use *HydraScalar*—our heavily modified version of *sim-outorder* from Wisconsin's SimpleScalar toolkit version 2.0 [19]—for our experiments. 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, targeting a portable, virtual instruction set (PISA) that most closely resembles MIPS IV [20]. The simulators instantiate a virtual machine and can emulate the object program's execution in varying levels of detail. Simulations do not model kernel behavior or context switches, instead performing operating-system calls by proxy; but otherwise all non-kernel behavior, including library code, is simulated.

HydraScalar simulates at the cycle level an out-of-order execution, five-stage pipeline: fetch (including branch prediction), decode (including register renaming), issue, writeback, and commit. We add three further stages between decode and issue to simulate time spent renaming and enqueuing instructions. Issue selects the oldest ready instructions for execution.

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 model mis-speculated paths. Like a real processor, HydraScalar checkpoints appropriate state as it encounters branches, and then proceeds down the predicted path, executing wrong-path instructions if appropriate. Upon detecting a mispredicted branch, wrong-path instructions are squashed, and recovery from the checkpointed state is straightforward. This modeling captures mis-speculation consequences like prefetching, cache pollution, and—if the predictor is updated speculatively—pollution in the branch predictor.

In our model, detecting mispredictions takes place in the writeback stage, and cleanup commences immediately, even though mispredictions may be detected out of order. Unlike misprediction handling, however, updating the predictor's state (history, up-down counters, etc.) takes place in-order, at instruction commit. The obvious exception are experiments that test early, speculative update of branch-history state.

Conditional-branch direction-mispredictions suffer at least a seven-cycle latency, because the branch condition does not resolve until the writeback stage. 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. In such a case, a BTB miss still redirects the fetch engine, but the detection of the misfetch during decode means the resulting bubble is only 2 cycles long. Indirect jumps, even though known to be taken, need to read the register file and HydraScalar assumes this action cannot be performed from decode. Indirect-jump targets therefore cannot be computed by the

Parameter	Value	Comments
Processor core		
RUU (register-update-unit) size	128	Instruction window
LSQ (load-store-queue) size	64	Enforces load-store ordering
Instruction register size	16 instructions	Buffer b/t fetch and decode
Decode/rename 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	any combination of 3 loads/2 stores	
Branch prediction		
Predictor style	gshare, PAg, or PAs	
BTB	2048-entry, 2-way	updated only if taken
Return-address stack	32-entry, repaired after mispredictions	repair: TOS ptr. & contents
Mispredict penalty	2 cycles for misfetch, 7 cycles otherwise	
Memory hierarchy		
L1 data-cache	128 K, 2-way (LRU), 32 B blocks, 8 MSHRs, 1-cycle latency	
L1 instruction-cache	128 K, 2-way (LRU), 32 B blocks, 1 cycle latency	
L2	unified, 8 M, 4-way (LRU), 32 B blocks, 4 MSHRs, 12-cycle latency	
Memory	200 cycles	
L1→L2 bus	1 transaction every 2 cycles	
L2→mem bus	1 transaction every 8 cycles	

Table 1: Baseline configuration simulated by HydraScalar.

dedicated adder in the decode stage, and if the BTB mispredicts the target, the error is only detected in the writeback stage. Since many entries in the direction predictor correspond to not-taken branches (or are simply idle), the BTB is decoupled [21], only allocating entries for taken branches. This permits the BTB to have fewer entries. The return-address-stack is updated speculatively, and repaired using the top-of-stack pointer & contents scheme [17].

The predictor we model makes a prediction for each branch fetched, but within a group of fetched instructions, those that follow the first predicted-taken branch are discarded, because control must now jump to a new location. This effectively means that the fetch engine fetches through not-taken branches but stops at taken branches. Each branch can potentially mispredict, requiring the processor to track some state to permit cleanup (for example, the register map, and in this paper, branch-predictor repair information). This *shadow state* can be tracked in the instruction window or in dedicated structures. This study explores the question of how to best organize this the shadow state.

Table 1 summarizes our baseline model. Since issue widths, instruction-window sizes, and so forth continue to grow, this work assumes an aggressive, 8-wide machine with a 128-entry instruction window and 128 Kbyte caches. The cache hierarchy is a conventional two-level, non-blocking organization with separate first-level instruction and data caches. The architectural registers (32 each for integer and floating-point) are separate and updated on commit; renaming determines whether operands reside in the RUU or in architectural state. A 64-entry load-store queue (LSQ) disambiguates memory references: stores may only pass preceding memory references whose addresses are known not to conflict. Finally, for these simulations, HydraScalar models a pipelined bus with a fixed fetch spacing.

2.2 Benchmarks

We primarily explore the SPEC95 integer benchmarks [18], summarized in Table 2. We use the provided reference inputs. All benchmarks are compiled using `gcc -O3 -funroll-loops` (`-O3` includes inlining). We also include three SPEC95 floating-point benchmarks, but because prediction accuracies are so high for the floating-point suite, we generally find that speculative update has little impact for them.

	Warmup Insts	Branches per Instruction				Branch Accuracies				Cond. Branch Counts	
		All	Return	Indir	Cond	All	Return	Indir	Cond	Dynamic	Static
go	926 M	0.144	0.011	0.002	0.111	0.725	1.000	0.629	0.656	5.6 M	3,973
m88ksim	26 M	0.212	0.018	0.003	0.162	0.957	1.000	0.251	0.955	8.1 M	231
gcc (cc1)	221 M	0.194	0.015	0.030	0.144	0.842	1.000	0.350	0.836	7.2 M	11,727
compress	2576 M	0.202	0.028	0.000	0.133	0.924	1.000	0.063	0.886	6.7 M	205
li (xlisp)	271 M	0.236	0.027	0.082	0.137	0.948	0.997	0.814	0.935	6.8 M	334
ijpeg	824 M	0.059	0.001	0.003	0.051	0.888	1.000	0.984	0.869	2.5 M	657
perl	601 M	0.193	0.019	0.077	0.129	0.937	0.990	0.332	0.954	6.5 M	352
vortex	2451 M	0.166	0.021	0.021	0.121	0.968	1.000	0.768	0.963	6.1 M	3,121
tomcatv	2276 M	0.194	0.000	0.000	0.131	0.999	na	na	0.999	6.6 M	51
hydro2d	376 M	0.262	0.000	0.048	0.180	0.998	1.000	0.689	0.998	9.0 M	251
mgrid	476 M	0.235	0.000	0.000	0.158	0.999	1.000	0.514	0.998	7.9 M	906

Table 2: Benchmark summary. Statistics are from the post-warmup, 50 M-committed-instruction simulation window, and use the baseline configuration in Table 1. “All” refers to all branches, whether conditional, direct-jump, indirect-jump, or return. “Indirect branches” do not include returns. “Branch accuracy” refers to target-address prediction, except for the conditional-branch column, which presents direction-prediction accuracies. These results are for a gshare predictor with 14 bits of global history and a 16 K-entry PHT that is updated in the commit stage.

Some benchmarks come with multiple reference inputs, in which case one has generally been chosen. For *go*, we choose a playing level of 50 and a 21x21 board with the `9stone21` input. For *m88ksim*, we use the `dhrystone` input; for *gcc*, `cccp.i`; for *ijpeg*, `vigo.ppm`; and for *perl*, we use the scrabble game. But for *xlisp*, we run the program with all the supplied LISP files as arguments.

2.3 Simulation Length

Running the SPEC benchmarks to completion with the “ref” inputs on a cycle-level simulator is prohibitive. For some benchmarks, each data point would take days. Using the shorter “test” or “train” inputs, on the other hand, risks unrepresentative results, because some of these inputs are simplistic or even trivial. Instead, we perform full-detail simulation for a representative, 50 million instruction segment of the program’s execution with the “ref” input. Cycle-level simulations are run in a fast mode to reach the chosen simulation window. In this fast mode no microarchitectural simulation takes place; only the caches and branch predictor are updated. Table 2 includes the length of the fast-mode (“warmup”) phase for each benchmark, including 1 million instructions in which simulation runs in full detail to prime other structures. Table 2 also shows for each benchmark the number of dynamic branch references seen during the simulation window, as well as the number of static branch sites seen. We compared the branch-site coverage for simulation windows of 50 million and 100 million instructions. For all except *go, gcc, tomcatv*, and *hydro2d*, the coverage is the same. For *go* and *gcc*, the 50 million instruction window covers 85% of the branch sites covered by the 100 million instruction window, and for *tomcatv* and *hydro2d*, the coverage is 50%.

Using a 50 million instruction simulation window reduces the simulation time substantially. Even if simulating each benchmark to completion took only one day, just producing the data in this paper would have taken 2.6 machine-years to complete! Instead, the experiments took only about 2 machine-months. Furthermore, unless context switches are modeled, simulating the benchmarks to completion is not necessarily more accurate, because it builds up branch predictor and cache state across the length of the program, behavior which unrealistically benefits branch-prediction and cache accuracy, more so than in the approach used here.

To ensure that our chosen segment does produce representative results we follow several steps. We use interval-miss-rate traces to verify that we have chosen a segment with representative branch-prediction, first-level data cache, first-level instruction cache, and second-level cache behavior. Then we use cycle-level simulations to further verify that overall IPC and the relationship among the aforementioned parameters is representative.

For each benchmark, we first use a simple miss-rate simulator that measures the cache miss rate and branch misprediction rate for each 1 million instruction interval, independent of the previous interval. We gather data for the entire program's execution, then identify a candidate simulation window and test its validity using cycle-level simulation. For a range of cache and branch-predictor configurations, we compare the program's IPC during the chosen 50 million instruction window to the program's IPC for a much larger, 250 million instruction window. In the case of *go*, we used a 500 million instruction window instead. This approach is indeed simulation-intensive, but the cost is a one-time cost that can be amortized over an arbitrary number of studies that use these benchmarks. We have used these simulation windows for tens of thousands of individual simulations.

We have found that the single, most important factor when sampling this way is to avoid the program's initial phases, which might exhibit unusual behavior. *Compress*, for example, exhibits very different behavior for its first 1.5 billion instructions. This is solely an artifact of the SPEC95 benchmark version of *compress*; during this initial phase, the program generates the data that it will subsequently compress or decompress. The branch misprediction rate during this phase is approximately twice as high as during the rest of the program. *Perl, vortex*, and *tomcatv* are other programs with markedly different initial phases, and most of the SPEC95 benchmarks exhibit some startup behavior.

Altogether, this approach verifies that the chosen simulation windows provide representative behavior in terms of branch prediction accuracy, cache performance, and overall IPC. The use of interval-miss-rate graphs ensures that we avoid grossly unrepresentative behavior. Then the comparison of IPC across multiple configurations gives us IPC *surfaces* that permit us not only to verify the IPC itself, but also the validity of the relationship among branch-predictor configuration, cache configuration, and IPC. In all cases, the simulation window eventually chosen matches the longer 250 million instruction window well in all regards, and IPC values are accurate to within less than 4% (usually within 1%).

The approach is described in more detail in [22].

3. Speculative Update for Global History

3.1 The Need for Speculative Update

Global-history predictors, like GAg[2] [1] and gshare [8], 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 incorporates branch-address bits. Gshare, for example, xor's 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.

[2] Global history, Adaptive PHT, Global PHT

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

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

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. Of course, 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.

Non-speculative update presents a second timing problem, described in detail by Hao *et al.* in [14]. 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. Even in-order processors exhibit variability if they use non-blocking caches. 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 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 branches.

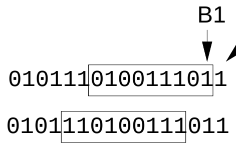

Figure 1: Two occurrences of branch B3 with the same actual history of preceding branch results. But different numbers of outstanding branches cause the history that appears in the global-history register (shown by the boxes) to vary.

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

3.2 Results

The prior discussion qualitatively illustrated the benefits of speculative update if implemented with a suitable fixup mechanism. The results in Figure 2 quantitatively demonstrate 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 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 as a function of global-history length for "full" gshare predictors of different sizes (*i.e.*, every bit of history is xor'd against a branch-address bit). For each configuration, the PHT contains $2^{\text{history-bits}}$ entries, and the processor allows 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 a mean of 10% for SPECint using a 16-bit/64k-entry 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* 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.

We also performed tests for GAg predictors, and found almost identical results.

Speculative Update with and without Fixup. In addition to the demonstrated benefit of speculative update, Figure 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 we therefore reduced our simulation requirements by limiting the number of data points gathered for no fixup.

Number of Outstanding Branches. Figure 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 gshare and a 16-bit global history but an infinite-sized PHT, to focus on the role of in-flight branches and avoid artifacts from PHT aliasing. 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. We nevertheless include these configurations to more fully map out the shape of the curves.

As expected, 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 largely flattens out at sufficiently high in-flight capacities of 4–8 branches for the integer programs. *Hydro2d* behaves similarly, although the effect is less pronounced, and it takes advantage of a larger in-flight capacity: the plateau occurs at 20 in-flight branches. *Tomcatv* and *mgrid* also take advantage of large in flight capacities, and as in the previous figure, are indifferent to the update timing.

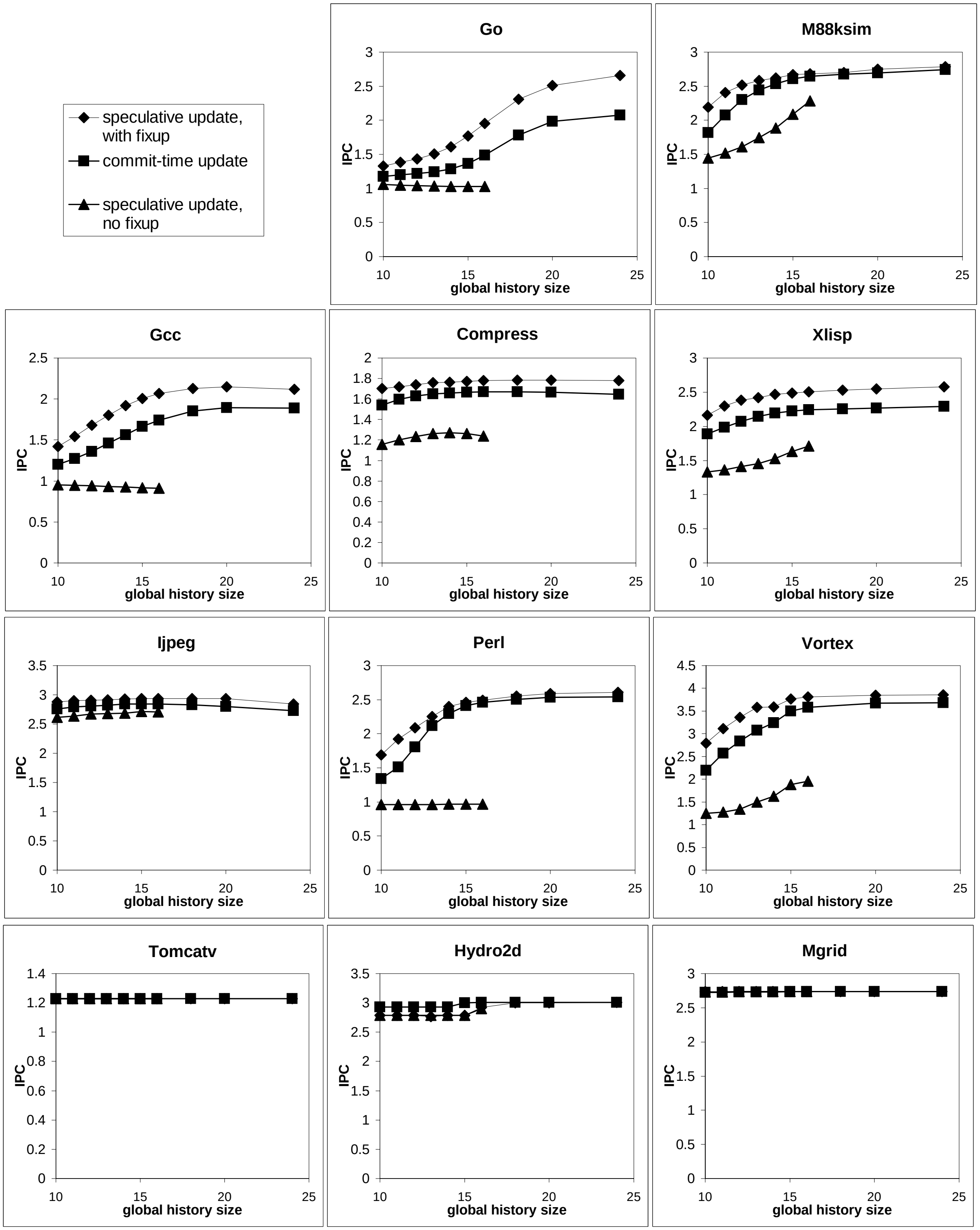

Figure 2: Performance of a “full” gshare branch predictor as a function of size and history-update policy. The x-axis shows global-history length, and the PHT contains $2^{\text{history-bits}}$ 2-bit counters. The curves coincide for tomcatv and mgrid. Results were obtained with 32 in-flight branches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

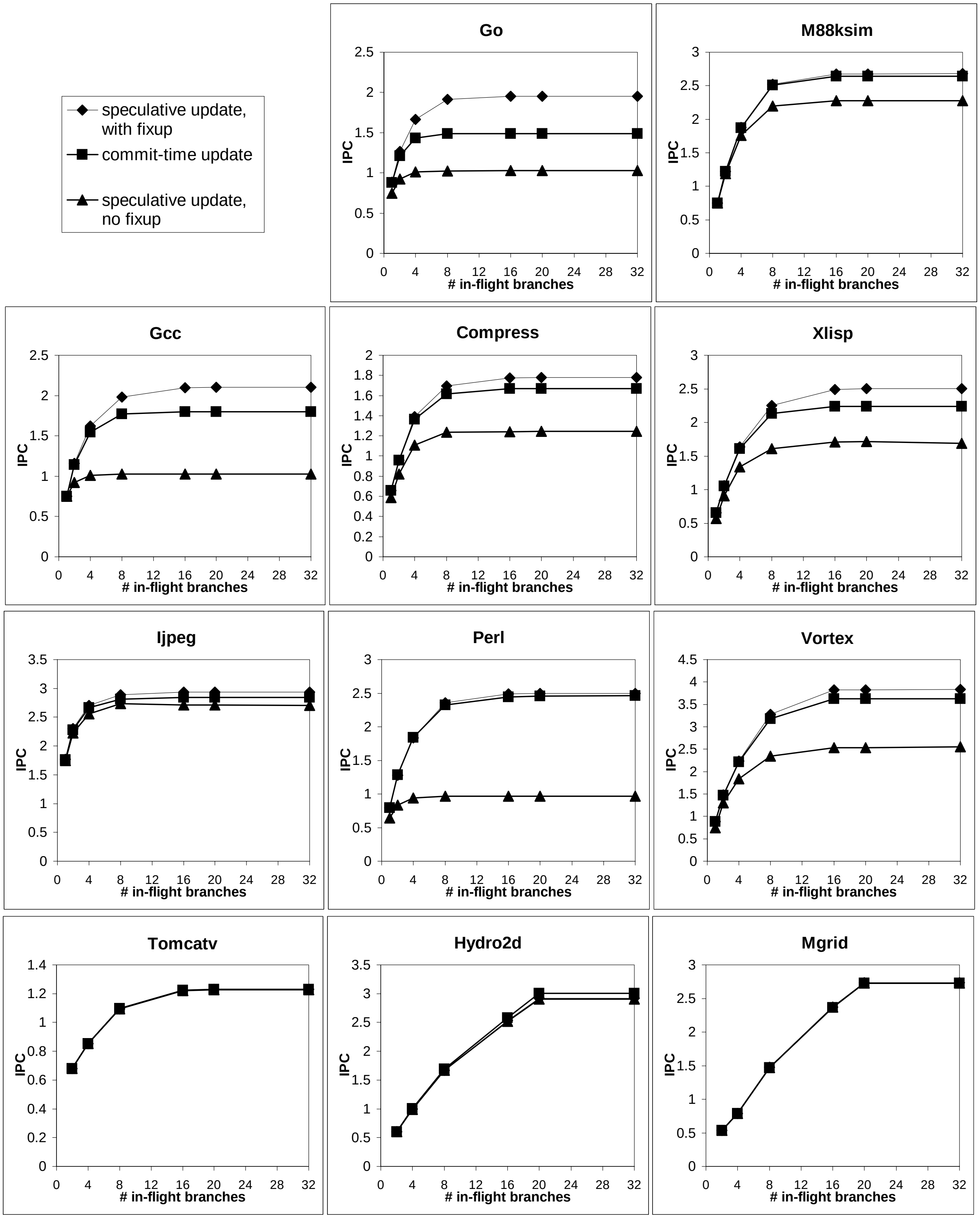

Figure 3: Performance of a gshare branch predictor as a function of the maximum number of in-flight branches (both conditional and unconditional) permitted in the processor. All results in these graphs were obtained with a 16-bit gshare organization, but with an alias-free PHT.

3.3 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 [15]. 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 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 register's 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.

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.* [15] 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 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 [11].

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-

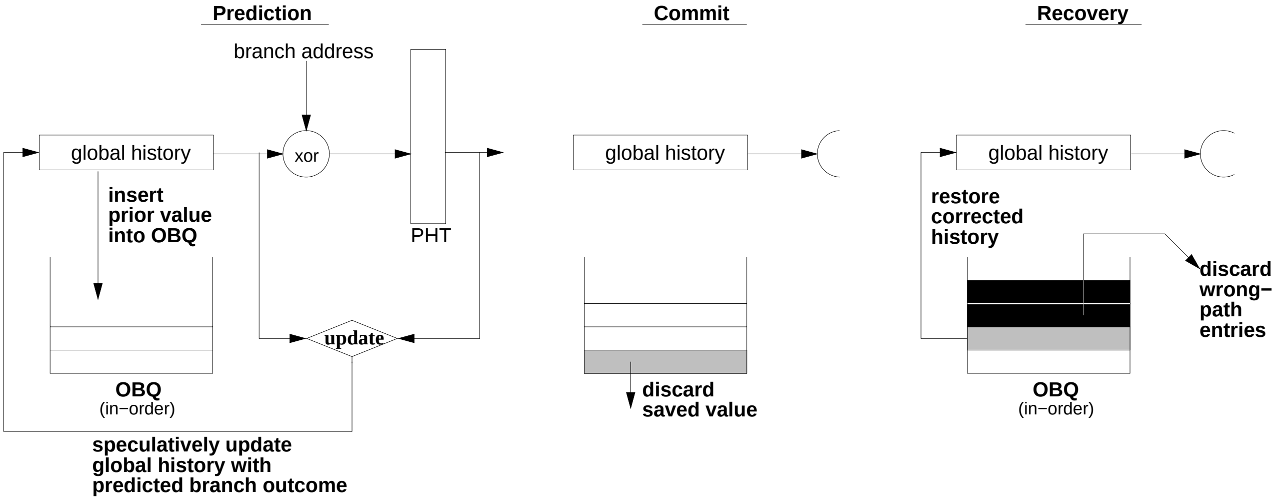

Figure 4: Checkpointing- or history-based fixup for a global-history predictor. On a prediction, the current global history is saved in the OBQ, then updated with the predicted branch outcome. At commit, the saved global history can be discarded. When recovering from a misprediction, wrong-path saved histories are discarded, and the history that existed just before the misprediction is restored and updated with the correct branch outcome.

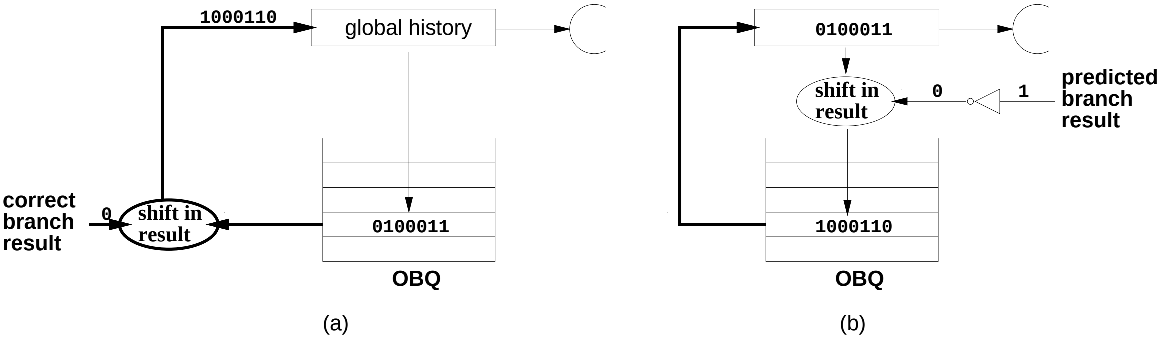

Figure 5: Two techniques for obtaining the corrected outcome at fixup. (a) The OBQ saves histories, and shifts in the correct branch result at misprediction-recovery. (b) The OBQ saves already-corrected histories; this works because the saved values are only used after a misprediction. Here the predicted 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.

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, discarding entries that follow the mispredicted branch, and updating with the correct outcome the OBQ entry associated with the mispredicted branch. 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.

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.

OBQ Size. An OBQ—whether used for the history- or the future-based technique—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 [11], 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), we choose not to further explore this truncated version of the OBQ.

4. Speculative Update for Local History

4.1 The Need for Speculative Update

Local-history based predictors like PAg[3] or PAs[4] use a table of history registers (the *branch history table*, or 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 local-history predictors unless branches happen to alias to the same history-table entry. Local history benefits branches that follow a reliable pattern, and especially those 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 [23], especially when mechanisms for reducing or eliminating aliasing in the BHT are assumed [24].

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-inaccurate global-history register. In a local-history scheme, on the other hand, a flaw in some BHT entry affects only the branches reading that particular entry. But multiple instances of the same branch instruction can be in flight at one time, and with commit-time update, the value read from the BHT can therefore be stale. As one simple example, consider a short loop with a short iteration count. Local prediction could ideally identify which iteration is the last. Yet 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.

[3] Per-address history, Adaptive PHT, Global PHT

[4] Per-address history, Adaptive PHT, Set-associative PHT

Our results in this section show that speculative update remains important for local-history predictors—in fact, sometimes more important than for global-history predictors. Furthermore, if updating 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 various BHT entries. These wrong-path branches are squashed, but squashing does not undo the wrong-path updates to the BHT, leaving ”extra”, corrupt 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.

4.2 Results

For a subset of the benchmarks, Figure 6 shows the differences among an idealized, PAs, local-history predictor (1) with speculative update plus fixup, (2) with commit-time update, and (3) with speculative update but no fixup, all as a function of BHT width. To avoid artifacts from BHT or PHT aliasing in these experiments, the BHT is infinitely deep and therefore alias-free, and the PHT is held fixed at 1 M entries. To see the effect of update timing on more realistic configurations, Figure 7 repeats the same experiments, but this time with a more realistic PAg organization consisting of a BHT containing 1 K entries and a PHT containing $2^{\text{history-bits}}$ entries. The 1 K-entry BHT was chosen to match the size of the BHT in the local-history component of the Alpha 21264’s hybrid predictor [10, 11].

Once again, the results make clear the importance of speculative update with repair. In most cases, speculative update matters less for local-history predictors than for global-history, but the superiority of speculative update with fixup is nevertheless clear. For *xlisp*, speculative update actually helps local history more than global history, and—when the BHT and PHT are both large and unlikely to suffer aliasing—speculative update also matters more for local than global for *m88ksim* and *perl*.

The next set of experiments explores the relationship between BHT depth and update timing. Figure 8 compares the same update schemes for different BHT depths, with a width of 20 history bits. As in Figure 6, the PHT size is held fixed at 1 M entries. (This means that the 20-bit-history values in Figure 6 are the same as the “Inf” BHT values in Figure 8. In addition, since speculative update without fixup has already been shown to perform terribly, we simply omit that curve from Figure 8.) It turns out that, for *m88ksim, xlisp*, and *perl*, the impact of speculative update is not at all sensitive to the number of BHT entries. This is no surprise, given those programs’ small static footprints. In contrast, update timing is sensitive to BHT size for *go* and *gcc*, which have larger static footprints.

Speculative update is more sensitive to the history width stored in the BHT—Figures 6 and 7. But speculative update is most sensitive to PHT size and the degree of PHT aliasing. This can be seen in the difference between the results for an idealized PAs (Figure 6) and for PAg (Figure 7). PHT size itself probably matters less than the degree of PHT aliasing, which is actually more severe in Figure 7 than necessary: the PHT is indexed only by the history string, without any anti-aliasing techniques. PHT aliasing corrupts the 2-bit-counter values and generates mispredictions, regardless of update timing. As aliasing is reduced, the consequent mispredictions are eliminated, and this exposes opportunities for update timing to make a difference.

It was mentioned earlier that a stale global history potentially affects all subsequent branches. But for a local-history predictor—even with 20 or 30 outstanding branches—only a subset of the BHT contains stale state. Branches using entries that do not correspond to these in-flight branches see completely up-to-date state. Predictions that use those entries are therefore unaffected by the choice of update timing. This effect explains why speculative update generally matters less with local-history predictors than with global-history. The same effect probably explains why update timing shows sensitivity to BHT depth for *go* and *gcc*. Small BHTs have more aliasing; some BHT entries are already corrupted by aliasing; and in those cases, update timing is irrelevant. Larger BHTs suffer less aliasing; update timing has more effect; and so late update is more harmful.

In Figure 6, the no-fixup case is especially sensitive to history width. This is most likely because, without fixup, any corrupted BHT entries stay corrupted until the invalid history bits are shifted out. As the history width grows wider, this takes longer; indeed, some BHT entries may never become clean if later mispredictions add additional corrupted history before earlier, corrupt bits can be shifted out. In Figure 7, on the other

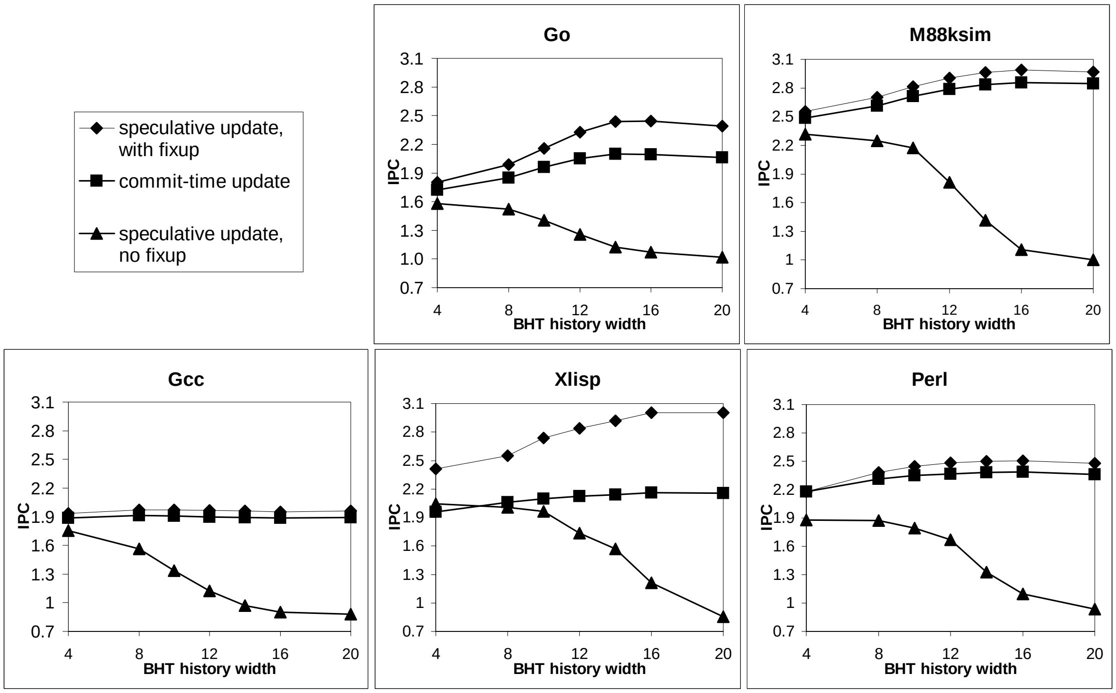

Figure 6: Performance of a PAs branch predictor as a function of BHT history width and history-update policy, for a BHT with infinitely many entries. The x-axis shows the history width. The PHT is held fixed at 1 M entries, regardless of BHT width.

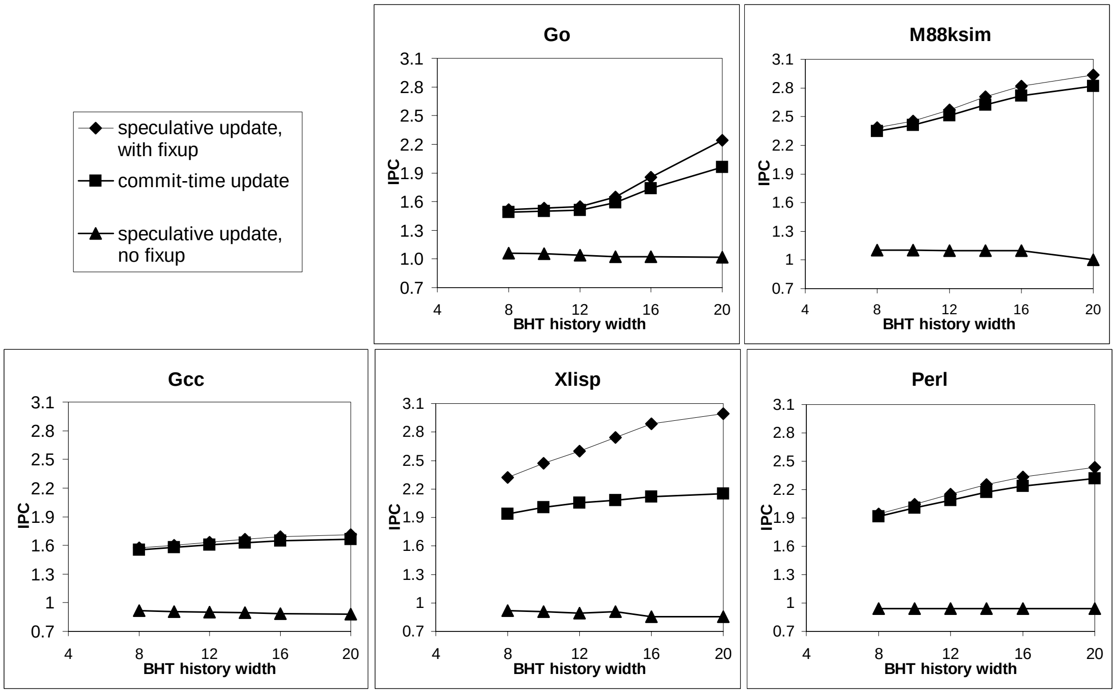

Figure 7: The same experiment as above, but for a PAg configuration consisting of a BHT with 1 K entries and a PHT with $2^{\text{history-bits}}$ entries. No data is presented for a history width of 4, because the PHT would only contain 16 entries.

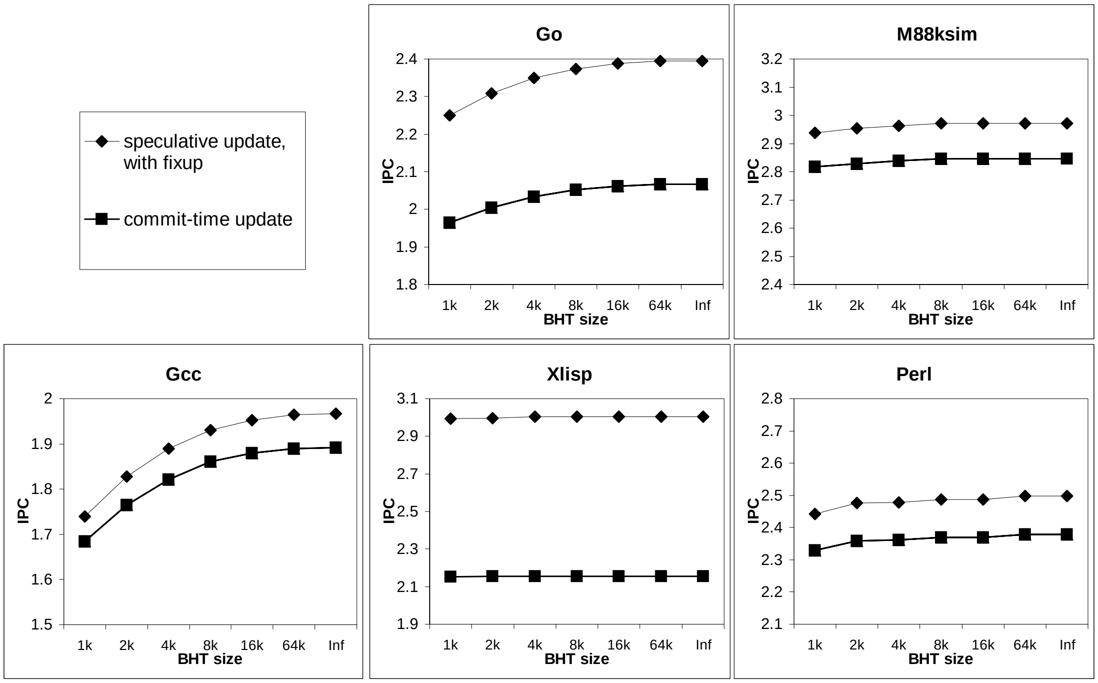

Figure 8: Performance of a PAs branch predictor as a function of BHT size and history-update policy, for a 20-bit-wide BHT. The x-axis shows the number of BHT entries; the PHT is held fixed at 1 M entries The trends are similar for 10-bit-wide BHTs, but as suggested by Figure 6, the gap between the curves is smaller.

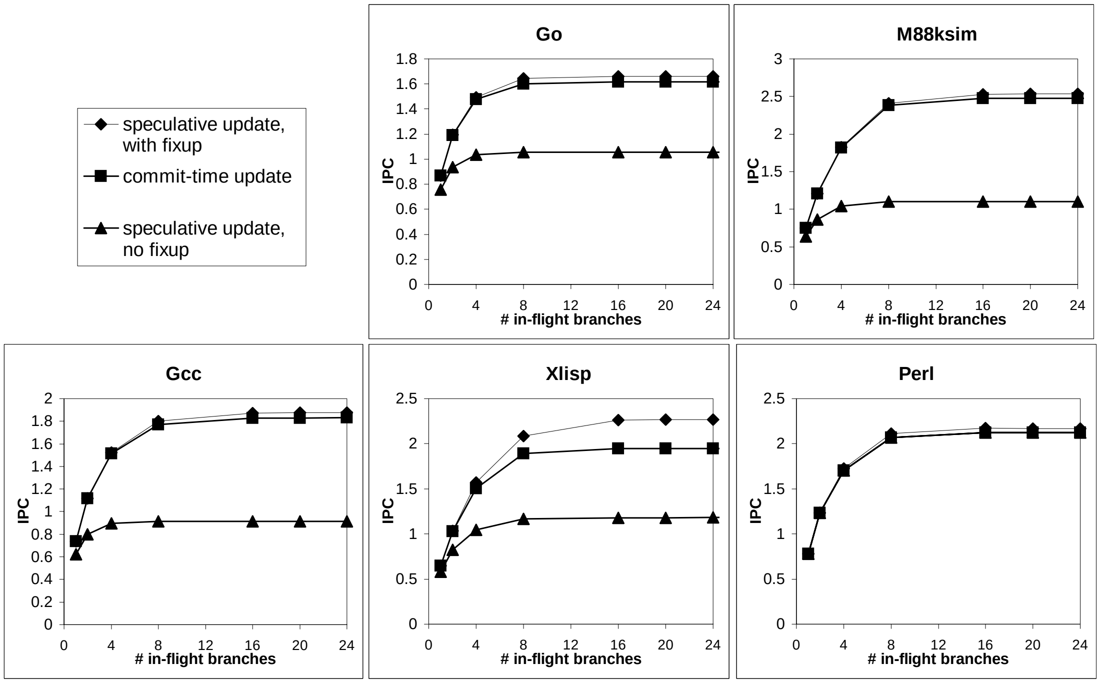

Figure 9: Performance of a PAg branch predictor as a function of the maximum number of in-flight branches permitted in the processor. The BHT is infinitely large and stores 10-bit histories. The PHT here contains 1 K entries. For *perl*, the curves for commit-time update and speculative update without repair coincide.

hand, the no-fixup case is consistently terrible. We again attribute this to the small BHT. Regardless of history width, the aliasing in the BHT means that many entries are likely to be indexed by more than one branch, and therefore likely to be corrupted by speculative update without repair.

Figure 9 shows the impact of different numbers of in-flight branches. As with the analogous global-history results, speculative update has more impact as the number of permitted in-flight branches increases. With commit-time update, the more branches in flight, the more out-of-date the history. Just as with global history, this trend reaches a plateau at 4–8 branches.

4.3 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 10 shows its operation.

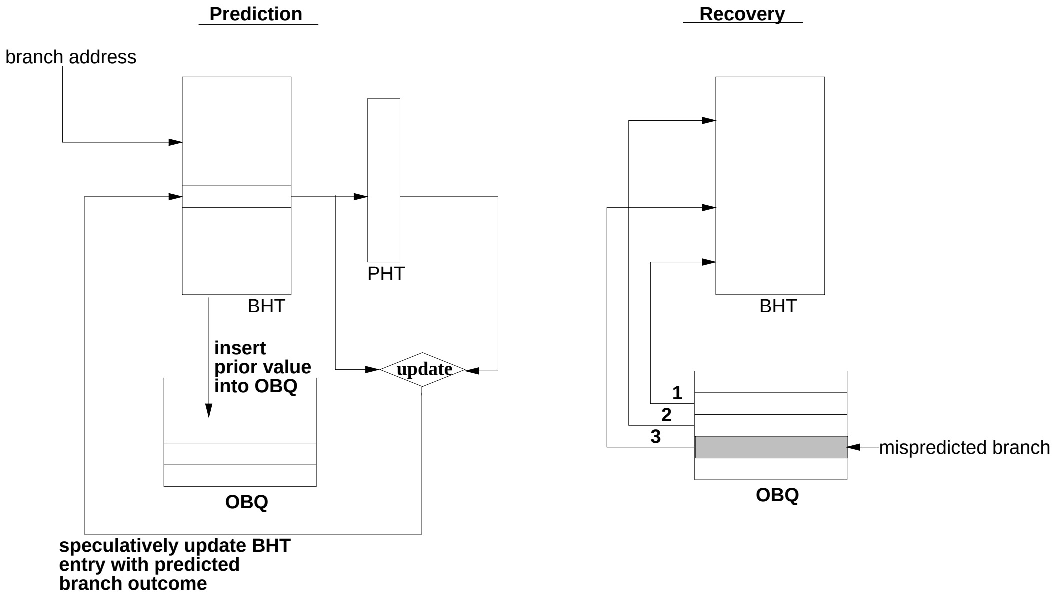

Figure 10: Repairing the BHT after a misprediction using a history file. The OBQ stores prior BHT entries that would otherwise have been overwritten. Recovery begins at the OBQ's tail and works forward to the mispredicted branch.

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.

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. (Only the oldest value needs to be replaced, but the history values need to be maintained in the proper order.) 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 [16].

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 becomes even more complex in terms of finding wrong-path branches among other reorder buffer contents, as every wrong-path branch must repair the state that it modified.

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 the oldest speculative history from the OBQ 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 11. The file has as many entries as the BHT, and is 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. Because 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 [16]. 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 (using the scheme described in [25, 26], for example), bloats the future file with extra bits and many comparators. Although space considerations pro-

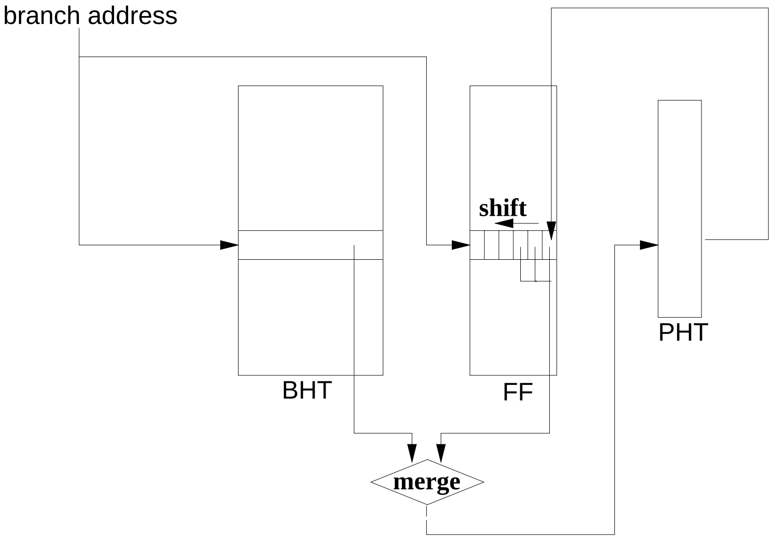

Figure 11: The prediction step using an "unordered" future file (FF). This diagram shows a future file that only stores speculative history bits, instead of entire history values. Here three bits are outstanding for the BHT entry in question. They are shifted left to accommodate the fourth bit which 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.

hibit our 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 12. As with global prediction, the OBQ must keep state for both conditional and unconditional branches.

We 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.

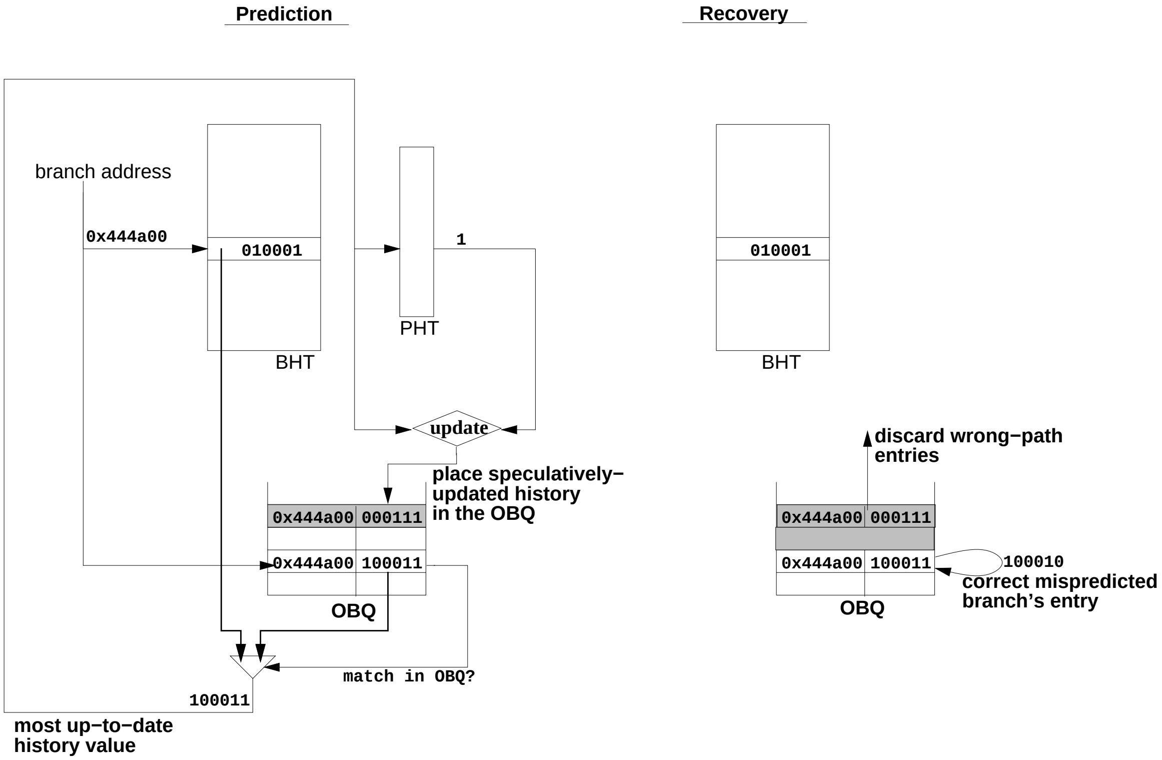

Figure 12: Repairing the BHT after a misprediction using an OBQ as a future file. The example shows a prediction (using the history value 100011 from the OBQ) of branch 0x4444a00. The new speculative history, 000111, is added to the OBQ. But the first instance of 0x444a00 was a misprediction; subsequent 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. 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. Most programs contain some branches that perform better with a global-history predictor and others that perform better with a local-history predictor [27], and even individual branches vary between needing global and local history [28]. So it is tempting to combine both a global- and a local-history predictor into a hybrid organization [8] that uses, for each branch instance, whichever predictor has recently been more accurate. 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 [12]. When one predictor is correct and the other incorrect, the counter value is either incremented or decremented, but does not wrap around (*i.e.*, like the saturating 2-bit counters in the PHT).

Because speculative update and suitable fixup are straightforward for global history, we consider in this section the impact of speculative update on the local-history component of a hybrid predictor. Our 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.

We consider two hybrid configurations. The first, loosely modeled after the Alpha 21264's hybrid configuration [10, 11], 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 an infinite number of 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.

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

Table 3: 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 16 K-entry PHT that uses approximately an equivalent number of bits as the hybrid predictor.

Table 3 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 [11]. The future file that local-history fixup requires can thus safely be omitted.

6. Conclusions and Future Work

This paper 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 we discuss practical hardware structures that implement perfect repair.
- 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, yet with speculative update and sufficiently wide histories, a more hardware-efficient local scheme outperforms even huge global-history predictors for some programs [23, 28].
- 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 com-

mitted 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 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 paper 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.

Acknowledgments

We would like to thank Matthew Farrens and the anonymous reviewers for their helpful comments. This work was supported in part by NSF grant CCR-94-23123, NSF Career Award CCR-95-02516 (Martonosi), and an NDSEG Graduate Fellowship (Skadron).

References

[1] 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*, pp. 124–34, May 1992.

[2] J. E. Smith, “A study of branch prediction strategies,” in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 135–48, May 1981.

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

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

[5] K. Diefendorff, “K7 challenges Intel,” *Microprocessor Report*, pp. 1, 6–11, Oct. 26 1998.

[6] K. Diefendorff, “Athlon outruns Pentium III,” *Microprocessor Report*, Aug. 23 1999.

[7] P. Song, “UltraSparc-3 aims at MP servers,” *Microprocessor Report*, pp. 29–34, Oct. 27 1997.

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

[9] L. Gwennap, “Intel’s P6 uses decoupled superscalar design,” *Microprocessor Report*, pp. 9–15, Feb. 16, 1995.

[10] L. Gwennap, “Digital 21264 sets new standard,” *Microprocessor Report*, pp. 11–16, Oct. 28, 1996.

[11] 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*, pp. 90–95, Oct. 1998.

[12] 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*, pp. 252–57, Dec. 1995.

[13] 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*, pp. 12–21, Apr. 1994.

[14] 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*, pp. 228–32, Nov. 1994.

[15] 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*, vol. 25, pp. 363–83, Oct. 1997.

[16] 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*, pp. 70–79, Dec. 1993.

[17] 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*, pp. 259–71, Dec. 1998.

[18] The Standard Performance Evaluation Corporation, "SPEC CPU95 Benchmarks." WWW site: http://www.specbench.org/osg/cpu95, Dec. 1999.

[19] 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 Department, July 1996.

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

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

[22] 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*, vol. 48, pp. 1260–81, Nov. 1999.

[23] 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*, pp. 257–66, May 1993.

[24] 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*, pp. 49–58, Dec. 1998.

[25] 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*, pp. 101–08, July 1998.

[26] 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*, pp. 250–59, July 1998.

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

[28] K. Skadron, M. Martonosi, and D. W. Clark, "Alloying global and local branch history: A robust solution to wrong-history mispredictions," Tech. Report TR-606-99, Princeton University Department of Computer Science, Oct. 1999.