Appears in the Proceedings of the 9th International Symposium on High Performance Computer Architecture

Reconsidering Complex Branch Predictors

Daniel A. Jiménez
Department of Computer Science
Rutgers University, Piscataway, NJ 08854
`djimenez@cs.rutgers.edu`

Abstract

To sustain instruction throughput rates in more aggressively clocked microarchitectures, microarchitects have incorporated larger and more complex branch predictors into their designs, taking advantage of the increasing numbers of transistors available on a chip. Unfortunately, because of penalties associated with their implementations, the extra accuracy provided by many branch predictors does not produce a proportionate increase in performance. Specifically, we show that the techniques used to hide the latency of a large and complex branch predictor do not scale well and will be unable to sustain IPC for deeper pipelines.

We investigate a different way to build large branch predictors. We propose an alternative predictor design that completely hides predictor latency so that accuracy and hardware budget are the only factors that affect the efficiency of the predictor. Our simple design allows the predictor to be pipelined efficiently by avoiding difficulties introduced by complex predictors. Because this predictor eliminates the penalties associated with complex predictors, overall performance exceeds that of even the most accurate known branch predictors in the literature at large hardware budgets. We conclude that as chip densities increase in the next several years, the accuracy of complex branch predictors must be weighed against the performance benefits of simple branch predictors.

1 Introduction

Accurate branch prediction is an essential component of today's deeply pipelined microprocessors. As improvements in process technology have continued to provide more transistors in the same area, branch predictors have become larger, more complex, and more accurate. This trend will continue into the future, with branch predictor hardware budgets running into the scores or hundreds of kilobytes. For example, the design for the Compaq EV8 included a hybrid branch predictor with approximately 45KB of state [14].

Figure 1 shows the arithmetic mean misprediction rates achieved over the SPEC 2000 integer benchmarks by extending several branch predictors from the literature into large hardware budgets, as well two very complex but highly accurate predictors: the multi-component hybrid predictor [5] and the perceptron predictor [8].

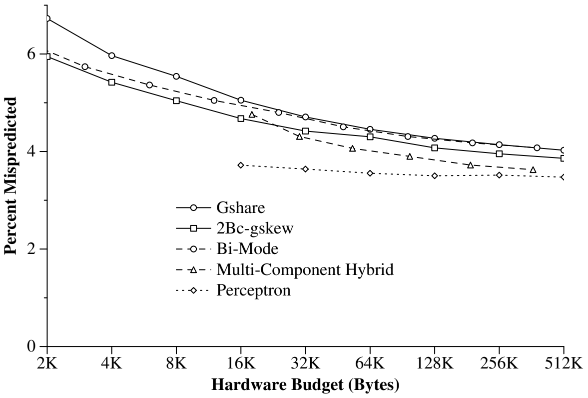

Figure 1. Arithmetic mean misprediction rates on SPECint 2000

1.1 Better Accuracy Doesn't Always Mean Better Performance

Unfortunately, a large hardware budget and complex organization can have a negative impact on performance relative to equally large but simpler branch predictors. Because the branch predictor is on the critical path for fetching instructions, it must deliver a prediction in a single cycle [7]. However, as pipelines deepen and clock rates increase, access delay significantly decreases the size and accuracy of large on-chip SRAM arrays such as branch predictors that can be accessed in a single cycle [1]. Since larger branch predictors are more accurate, modern microarchitectures attempt to overcome the access delay problem by using complex schemes such as hierarchical *overriding* branch predictors, in which a simple and quick predictor is backed up by a large, complex, and slower predictor [7]. This technique was used for the Alpha EV6 branch predictor [9] and was built into the design for the Alpha EV8 branch predictor [14]. Because of penalties associated with their implementations, the extra accuracy provided by such complex predictors does not yield a proportionate increase in performance.

To illustrate our point, Figure 2 shows the instructions per cycle (IPC) rates yielded by the multi-component hy-

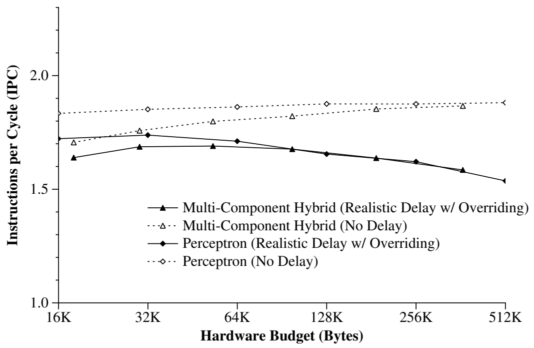

Figure 2. Ideal IPC for perceptron and multi-component predictors, contrasted with realistic, overriding versions

brid predictor and the perceptron predictor. We show the IPCs yielded by ideal, zero-delay versions of these predictors, as well as IPCs yielded by more realistic versions that use overriding. Overriding has been shown to yield better performance [7] than other proposed delay-hiding schemes such as lookahead [21] and cascading [7, 4]. However, no study of which we are aware has addressed the utility of overriding or other techniques on very large hardware budgets in the hundreds of kilobytes. Figure 2 shows that, *even with the best known delay-hiding techniques*, large branch predictors lead to significantly worse overall performance than more moderate sized predictors. For instance, the 512KB version of the perceptron predictor yields an IPC 11% less than that of the 32KB version, even though the larger predictor is more accurate. This is because overriding cannot completely hide the access delay of the branch predictor; when the slow and fast predictor disagree, a penalty proportional the the predictor access delay must be paid. Obviously, no microarchitect would actually design such a large branch overriding branch predictor with such a negative impact on performance. We conclude that either branch predictors cannot get larger than a few tens of kilobytes, or we must find another predictor organization capable of completely hiding access delay.

1.2 Our Solution

We propose an alternative design that eliminates the problems of previous delay-hiding organizations. We describe a simple, large branch predictor that can be pipelined to deliver every prediction in a single cycle, thus solving the problem of access delay. We present the design of our branch predictor and contrast it to previous work, and we present the results of simulations showing that our simplified predictor is capable of improving performance over the best known branch predictors in the literature. We make optimistic assumptions about the implementations of the complex predictors we study, providing an upper bound on the accuracy of predictors that may actually be designed by microarchitects. We conclude that in the next several years, as hardware budgets allow for larger branch predictors, the ability to pipeline a branch predictor will become a more important design criterion than a clever prediction scheme that improves accuracy incrementally.

1.3 The Basic Idea

Our idea is to pipeline the branch predictor so that it always produces an accurate prediction in a single cycle. We apply our technique to a special version of the *gshare* branch predictor which we call *gshare.fast*. Because of its simple design, *gshare.fast* can be pipelined to provide a branch prediction in a single cycle. The key idea is that our predictor is organized so that a small set of candidate entries from the pattern history table (PHT) is prefetched several cycles before the prediction is needed. Figure 3 illustrates the notion of narrowing down the number of entries from which to select the final prediction. As more instructions are fetched and executed, the location of the exact PHT entry needed to make a prediction becomes known, and by the time branch prediction is required, the prediction can be selected in a single cycle. This idea can be applied to any simple predictor that uses only global history and a few bits from the address branch PC. The predictor can produce a prediction in a single cycle because, unlike complex predictors, it does not use local histories, compute hashing functions, or otherwise create dependences between the branch address and the prefetched PHT entries.

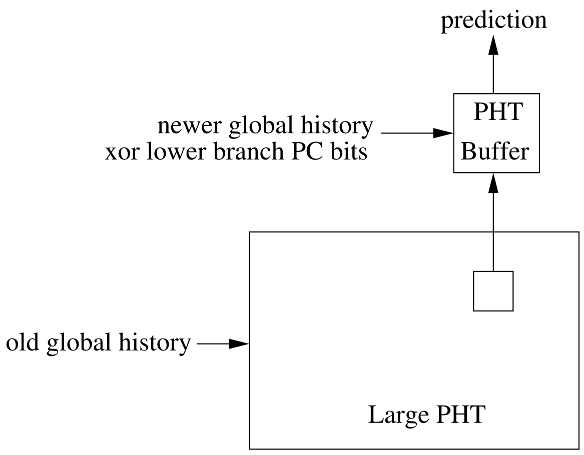

Figure 3. *gshare.fast* **separates the process of indexing the PHT into stages, prefetching a contiguous set of candidate predictions.**

This paper is organized as follows. We first present some background into branch prediction, describing some basic concepts and reviewing some implemented and proposed complex branch predictors. We then discuss the impact of branch predictor complexity on performance. We describe our solution to the problem, giving a detailed discussion of *gshare.fast*. We provide experimental results showing the advantages of our predictor and the conclude.

2 Impact of Branch Predictor Complexity

Branch predictor complexity leads to branch predictor delay, which has a large negative impact on performance [7]. Techniques have been proposed and implemented that hide some of the delay, but none of these techniques can hide the delay completely and still maintain predictor accuracy. Thus, branch predictor delay will always remain a problem for complex predictors. In this section, we describe some existing branch predictors, discuss the impact of complexity on branch predictor design, and review the solutions that have been proposed for this problem.

2.1 Some Implemented Branch Predictors

As microprocessor designs become more aggressive with deeper pipelines and higher clock rates, branch predictor designs have become more complex. We review a few of the branch predictors in industrial designs, in chronological order of their introduction. As we will see, industrial branch predictor designs have become more and more complex in recent years:

- The Alpha EV6 core uses a hybrid branch predictor composed of two two-level predictors [9, 20]. A 4K-entry PHT is indexed by a global history register while a 1K-entry PHT is indexed by one of 1024 local 10-bit branch histories. The final branch prediction is chosen by indexing a third chooser PHT. its implementation complexity comes with a cost. The Alpha branch predictor overrides a less accurate line predictor, introducing a single-cycle bubble into the pipeline whenever the two disagree [9].

- AMD's new Hammer (K8) architecture uses a sophisticated branch prediction scheme designed for large workloads [19]. An array of *branch selectors* cached in the L2 cache decides whether to use two-level prediction for hard-to-predict branches or to simply predict a branch bias for highly biased branches. Hard-to-predict branches are predicted by a 16K-entry pattern history table indexed by a global history register. Because of the complexity of the branch predictor, a predicted fetch introduces a single cycle bubble into Hammer's 12-stage pipeline [19].

- Seznec *et al.* describe the Compaq EV8 hybrid predictor, which has four components: two two-level predictors, one bi-modal predictor, and one meta-predictor [14]. The hardware budget for this predictor is 352 Kbits.

2.2 Sources of Branch Predictor Complexity

Branch predictor complexity comes from predictor organizations that place many levels of logic on the critical path to making a prediction. Some examples of this complexity are:

- Large tables indexed by the branch PC. If the branch PC or previous branch target address is needed to fetch a prediction from a large table with a high access delay, it might arrive too late and require overriding or some other latency hiding trick.

- Computation. Hybrid predictors must bring data from several tables together to compute a final prediction. This adds more gate delay to the process of making a prediction. Other predictors perform more complex computations. For instance, the perceptron predictor [8] computes the dot product of two vectors using a deep circuit similar to a multiplier.

- Depending on the contents of the branch instruction. Some architectures expose details of the branch predictor to the ISA. This idea puts the instruction cache on the critical path to making a prediction.

2.3 Branch Predictor Delay

Instruction fetch bandwidth is a bottleneck in modern microprocessor designs. Thus, it is critical that branches be predicted as quickly as possible to avoid introducing fetch bubbles into the pipeline. As explained in the Introduction, eliminating branch predictor access delay is essential to sustaining instruction fetch rates and thus performance.

2.3.1 Table Access Delay

As with caches, larger prediction tables take longer to read. The problem is particularly bad with branch predictors because of the increased decoder delay. Consider two SRAM tables of 4KB: a direct-mapped L1 data cache with 32-byte lines, and a pattern history table with two-bit saturating counters. An access to the cache requires selecting among only 128 entries, while the pattern history table requires selecting among 16K entries. Thus, a pattern history table incurs a higher decoding cost than a cache with the same size.

2.4 Increased Predictor Complexity

Complexity also contributes to delay. In addition to table accesses, these predictors incur extra delay by, for instance, combining information from multiple tables, using an index read from one table to access another table, computing information such as the majority of two counters [14], or even computing the dot product of a two vectors [8].

2.5 Impact of Delay

Jimenez *et al.* point out that a 100% accurate predictor with a latency of two cycles results in worse performance than a relatively inaccurate predictor with single-cycle latency [7]. Because of increasing clock rates and wire delay, the maximum size of a pattern history table accessible in a single cycle in future designs will be 1024 entries. Several techniques for mitigating branch predictor delay are discussed. Nevertheless, our research shows that *even with these techniques*, branch predictor delay still has a significant on performance.

2.6 Techniques for Mitigating Delay

2.6.1 Overriding Predictors

An overriding predictor uses two branch predictors: a quick but relatively inaccurate predictor, and a slower, more accurate predictor. When branch prediction is needed, the quick predictor makes a single-cycle prediction, after which instructions are fetched and speculatively executed. A few cycles later, the slower predictor returns its prediction. If the two predictions disagree, the few instructions fetched and issued are squashed, and fetch continues down the other path. This process incurs a penalty smaller than a full branch misprediction. The overriding technique is used in the Alpha EV6 and EV7 cores, and is included in the EV8 design [9, 14]: a quick instruction cache line predictor is overridden by a two-cycle hybrid branch predictor.

There are two drawbacks to the overriding idea. First, although a high-latency predictor may have a higher overall accuracy, the improvement in performance is not directly proportional the the improvement in accuracy, since pipeline bubbles result from those cases when the quick and slow predictors disagree. Thus, the actual performance improvement is a function of the accuracies of the quick predictor and the slow predictor, as well as the latency of the slow predictor. Second, implementing the overriding mechanism introduces extra complexity into the design of the rest of the pipeline. Instead of having the two cases of correct and incorrect prediction, there are now four combinations of correct and incorrect quick and slow predictions. The processor designer must design logic to squash instructions for the case when the slow predictor overrides the fast predictor, and for when the overriding prediction itself is incorrect.

2.6.2 Dual-Path Fetch

An alternative to overriding is dual-path fetch, which is used in the AMD Hammer architecture [3]. When an instruction cache line is fetched that contains a branch instruction, it may be several cycles before the branch prediction for that instruction becomes available. During this time, the processor fetches and speculatively issues down both possible paths. This technique effectively halves instruction fetch bandwidth and available execution resources while the branch prediction is being computed, and is not scalable to situations where multiple branches are being predicted in the front end.

3 Our Solution: A Large, Fast *gshare*

In this section, we present a highly accurate large branch predictor based on *gshare* [10] called *gshare.fast* that produces its prediction in a single cycle, uses the most recently updated history, and incurs no added penalty or complexity for the rest of the pipeline. We first describe the idea for predicting a single branch per cycle using a pipelined *gshare*, then discuss related issues such as a way to extend the idea to predict multiple branches in a single cycle.

3.1 A Pipelined Implementation

Figure 4 shows a diagram of a pipelined implementation of *gshare.fast*. Note that this pipeline runs in parallel with the rest of the fetch engine components, interacting with the components only to receive a branch address, send a prediction, or recover from a misprediction. There is no overriding or other overhead. Our results model several different PHT access latencies, but for this discussion we will assume that the PHT has an access latency of three cycles, and can return one line of 8 two-bit counters on each cycle. The branch predictor is pipelined in four stages: three for reading the PHT, and one for computing a prediction. In order to keep track of new speculative global history encountered since the PHT access began, the first three stages of the predictor pipeline each contain four latches. The first latch, which we call the *Branch Present* latch, records whether a branch was fetched and predicted during the corresponding cycle. The second latch, which we call the *New History Bit* latch, records the corresponding speculative global history bit (if any) shifted in from the next predictor pipeline stage. The third and fourth latches (not pictured) receive the corresponding bits from later pipeline stages in the first half of the cycle. To predict a branch fetched during cycle t, the predictor pipeline is organized as follows:

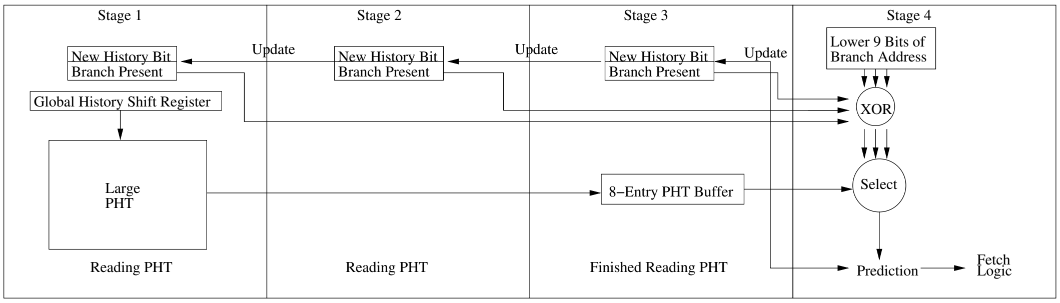

Figure 4. *gshare.fast* **pipeline for predicting a branch fetched at cycle** t

Stage 1, cycle $t-3$ In this cycle, the global history register is used to begin fetching a line of 8 two-bit counters from the PHT into the 8-entry PHT buffer. If a branch had been fetched and predicted during pipeline stage 2, the resulting New History Bit is shifted in from stage 2 and the Branch Present latch is set; otherwise, the Branch Present latch is reset.

Stage 2, cycle $t-2$ During this cycle, the multi-cycle PHT access continues. In the first half of the cycle, any history and Branch Present bits are forwarded to the previous stage. In the second half of the cycle, any New History and Branch Present bits are shifted in from stage 3 to maintain the current speculative global history.

Stage 3, cycle $t-1$ At the end of stage 3, the 8-entry PHT buffer has been read from the PHT. Any New History and Branch Present latches are forwarded to the previous stage in the first half of the cycle. In the second half of the cycle, New History and Branch Present bits are shifted from stage 4.

Stage 4, cycle t If a branch is fetched in this stage, the lower nine bits of its address are exclusive-ORed with the low bits of the global history register, shifted left and combined with up to three newly generated New History Bits from the previous stages. This computation forms an index into the 512-entry PHT buffer, selecting an entry whose value provides a branch prediction that is forwarded to the rest of the fetch engine. This stage sends New History and Branch Present bits to the previous stage.

3.2 Predictor Update

Here, we describe our mechanisms for updating the branch predictor. There are three types of update:

- Speculative update of the global history register. When a branch is predicted, the global history is updated speculatively to reflect this prediction, on the assumption that the prediction is correct. This policy has been shown to produce better accuracy [17]. To speculatively update the global history in our predictor, the New History Bit and Branch Present latches are shifted to previous pipeline stages as described above. A global history shift register records history bits that are shifted out of the first predictor pipeline stage.
- Non-speculative update of the PHT. When a branch is executed, the PHT entry that provided the prediction for that branch must be incremented or decremented based on whether the branch is taken or not taken. Because it takes several cycles to update the predictor, a problem occurs when in-flight branches need access PHT entries that have not yet been updated. Our approach is to simply update the table slowly. This policy has a negligible impact on predictor accuracy since the PHT entry for a particular address/history combination tends not to change very much. For instance, in our simulations, we have found that if we allow the branch predictor to predict 64 branches between the time it predicts and updates any branch, the average misprediction rate for a 256KB budget increases from 4.03% to 4.07%, with less than 1% decrease in IPC.
- Recovery after a misprediction. When a branch misprediction occurs, the speculative global history register must be overwritten with a non-speculative history that is updated only when branches are executed. The information in the PHT buffer will be invalid for the few cycles it takes to refill it. To overcome this latency, the PHT buffer can be checkpointed to a buffer associated with the pipeline stage where branch prediction occurs. This buffer is propagated to similar buffers kept for each pipeline stage. During a misprediction, the buffer associated with the commit stage, whose contents reflect the PHT entries associated with the by now non-speculative global history, is copied into the PHT buffer. The copies of the PHT buffer that had been fetched and stored for previous pipeline stages will serve to fill the PHT buffer until the *gshare.fast* pipeline is filled again. The portion of the history register that had been used to fetch these copies contained

the correct and now non-speculative history, since any speculative and wrong-path histories recorded in the global history register would only have been used in indexing the PHT buffer, not fetching from the PHT itself. This idea requires no interaction between the microarchitecture and branch predictor, since there is no communication between the buffers and the pipeline stages with which they are associated.

3.3 Discussion

The ideas we introduce with our predictor raise several issues. We discuss some of them here.

3.3.1 Multiple Branch Prediction

We have presented our branch predictor in the simple setting of a fetch engine that predicts at most one branch per cycle. However, modern wide-issue microarchitectures often need to predict many branches at once. Methods for doing multiple branch prediction have been proposed [21, 15]. For instance, the EV8 branch predictor is capable of predicting as many as 16 branches from two fetch blocks in the same cycle [14]. It does this by arranging the dual-ported branch prediction table such that the branch predictions for each fetch block reside in consecutive locations in the table, so that they can be fetched all at once. This predictor uses stale speculative global histories that can be as many as three fetch blocks old. Using these stale histories is reported to have minimal impact on branch prediction accuracy.

Our predictor design can also be enhanced to provide many predictions in a single cycle. Predictions for consecutive branch instructions are already laid out close to one another in the PHT buffer. By enlarging this buffer, more branches can be predicted simultaneously. Suppose the latency of the PHT is k cycles. The PHT buffer has to be large enough to accommodate each combination of PHT entries that might be required after k cycles. In our original design where we predict up to one branch per cycle, the PHT buffer size can be as small as 2^k entries. If we enhance the design to accommodate p predictions per block, then the PHT buffer size should increase to $2^k p$ entries. For example, if up to 8 branch instructions can be fetched in a single cycle and the branch predictor latency is 3 cycles, then the PHT buffer must have at least 64 entries. The Branch Present and speculative history latches must also be increased in each pipeline stage by a factor of p, since up to p new speculative history bits may be generated in each cycle. This organization can still be easily designed to provide single-cycle access to the branch predictor. If necessary, an extra predictor pipeline stage can be introduced to separate the process of reading the PHT buffer from the process of computing the index with which it is indexed.

3.3.2 Large Hardware Budgets

Currently, branch predictors with hardware budgets in the hundreds of kilobytes are still considered infeasible due to limits on the number of transistors available on a chip. However, even now, a PA-RISC processor has been built with over 2MB of on-chip cache [18]. Intel has recently announced that it has built a chip only 109 square millimeters in size with 52 megabits of SRAM in its next-generation 90 nm process technology [13]. Thus, it is reasonable to expect that a 100KB branch predictor consuming less than 2% of the area of the chip will be designed for future processor generations.

3.3.3 Branch Targets

This paper is concerned only with the design of the branch direction predictor, which predicts whether a conditional branch will be taken or not. The branch direction predictor is a component of the branch target predictor, which predicts the address of the next instruction to fetch after a branch. The branch target predictor can be implemented as a branch target buffer (BTB). If a branch is predicted to be taken, the branch target predictor must predict the branch target so that instruction fetch may continue along the predicted path. Our predictor is useful even in situations with a slow BTB because the branch target only needs to be predicted if the branch is predicted taken, and compilers try to minimize the number of taken branches to avoid the penalty of a discontinuous fetch.

3.3.4 Interactions With Microarchitecture

We have claimed that our predictor reduces design complexity in two ways: it is conceptually simple, and it minimizes interactions with other aspects of the microarchitecture. This second point is important: the only interactions between the rest of the microarchitecture are when the fetch engine requests and receives a prediction, and when the microarchitecture signals the predictor to recover from a misprediction. We contrast this with the interactions with an overriding predictor, such as the ones in the Alpha EV6 and EV8 cores that require the interactions mentioned as well as the following:

- When the quick and slow predictors disagree, the few instructions fetched so far must be squashed and instruction fetch restarted down the path indicated by the slow predictor.
- The microarchitecture must keep track of two types of speculative instructions: instructions predicted with the quick predictor that must be squashed when the slow predictor overrides them, and instructions predicted with the slow predictor. Even when the quick

and slow predictors agree, the instructions predicted with the quick predictor must be re-marked as having been predicted with the slow predictor.

4 Experimental Results

In this section, we present experimental results showing that our *gshare.fast* predictor achieves better performance than the best branch predictors in the literature. We describe our experimental methodology, discuss the accuracies of each predictor, and present performance results measured with instructions-per-cycle (IPC).

4.1 Experimental Methodology

4.1.1 Predictors Simulated

We use the perceptron predictor [8], multi-component hybrid predictor [5], and the *2Bc-gskew* predictor [11]. The perceptron predictor and multi-component hybrid predictor are the most accurate known branch predictors in the academic literature. The recently presented EV8 branch predictor is a practical implementation of *2Bc-gskew* [14]. For each predictor, we use the overriding mechanism to mitigate access delay. For the perceptron predictor, we use both global and local history as input to the predictor.

4.1.2 Optimistic Assumptions

Our claim is that *gshare.fast* provides superior performance to these other predictors. To strengthen our claim, we choose optimistic assumptions for the other predictors. Thus, our results provide an upper bound on the performance achievable by more realistic, stripped-down versions of these complex predictors. We make the following assumptions:

Speculative Update Both global and local history are updated speculatively and recovered with no latency after misprediction. This policy has been shown to give the best accuracy, but is difficult to implement, especially with large tables [17].

Latency Most of the access latency for the table based predictors comes from the time it takes to select the counters from the various tables, with only one fan-out-of-four inverter gate delay. For the perceptron predictor, we assume that the delay in computing the perceptron output is a single cycles, and that the rest of the delay comes from the table lookups. Although the actual implementation of these predictors would likely have a higher latency, even a more practical complex predictor design, the branch predictor of the Alpha EV8, has a two-cycle latency, the same as the multi-component predictor we model for up to a 53KB hardware budget.

Parameter	Configuration
L1 I-cache	64 KB, 64-byte lines, direct mapped
L1 D-cache	64 KB, 64-byte lines, direct mapped
L2 cache	2 MB, 128-byte lines, 4-way set assoc.,
BTB	512 entry, 2-way set-assoc.
Issue width	8
Pipeline Depth	20

Table 1. Parameters used for the simulations

Overriding Penalty The cycle penalty associated with overriding a quick prediction is equivalent to the access latency of the branch predictor, with no extra time taken to squash instructions or fetch from the other path.

Clock Rate We assume an aggressive clock period of 8 fan-out-of-four inverter (FO4) delays, yielding a 3.5 GHz clock rate in 100 nm process technology. Hrishikesh *et al.* suggest that 8 FO4s is the optimal clock period – 6 FO4 for doing useful work and 2 FO4 for latch delay – giving the best combination of pipeline depth and time for useful work in the processor [6]. Jimenez *et al.* show that the largest pattern history table accessible at this clock rate contains 1K entries [7]. However, we optimistically assume that the quick predictor in the overriding scheme can contain 2K entries.

4.1.3 Execution Driven Simulations

We use the 12 SPEC 2000 integer benchmarks running under a modified version of SimpleScalar/Alpha [2], a cycle-accurate out-of-order execution simulator, to evaluate our branch predictors. To better capture the steady-state performance behavior of the programs, our evaluation runs skip the first 500 million instructions, as several of the benchmarks have an initialization period (lasting fewer than 500 million instructions), during which branch prediction accuracy is unusually high. Each benchmark executes over one billion instructions on the `ref` inputs before the simulation ends. Table 1 shows the microarchitectural parameters used for the simulations.

4.1.4 Branch Predictor Configuration

For the multi-component hybrid and perceptron predictors, we use the configurations of global and local history lengths and table sizes reported for the corresponding hardware budgets in the literature [5, 8]. Our *gshare.fast* predictor uses the maximum history length possible, i.e., the base-two logarithm of the number of PHT entries. We explore hardware budgets containing powers of two entries in the range of the hardware budgets chosen for the other two predictors.

4.1.5 Estimating Predictor Access Delay

Each predictor has two delay component: PHT access latency and computation time. For the multi-component and *2Bc-gskew* predictors, we estimate the latency of the largest table component and optimistically add a single fan-out-of-four inverter delay for the computation cost, e.g., for *2Bc-gskew*, the cost computing the majority function and choosing between the bimodal table and the *gskew* predictor. In addition to a table access, the perceptron predictor has a substantial computation component, estimated to be at least two cycles at our clock rate [8]. We optimistically assume that, with clever custom logic design, the perceptron predictor will take only one cycle to compute a prediction once the table has been accessed.

We estimate pattern history table access times for 100 nm process technology using the CACTI 3.0 [16] tool for simulating cache delay. This modified version of CACTI is more accurate than the original in several ways. First, while the original version of CACTI 2.0 [12] uses a simplistic linear scaling for delay estimates, the modified simulator uses separate wire models to account for the physical layout of wire interconnects: thin local interconnect, taller and wider wires for longer distances, and the widest and tallest metal traces for global interconnect. Second, wire resistance is based on copper rather than aluminum material properties. Third, all capacitance values are derived from three-dimensional electric field equations. Finally, bit-lines are placed in the middle layer metal, where resistance is lower.

Table 2 shows the access delays we computed for the multi-component hybrid predictor, *2Bc-gskew*, and the perceptron predictor at several hardware budgets, using the optimistic assumptions mentioned in Section 4.1.2 and a clock period of 8 fan-out-of-four inverter delays.

4.2 Misprediction Rates

Figure 5 shows the misprediction rates of the four large predictors simulated over hardware budgets ranging between 16KB and 512KB. We report arithmetic mean misprediction rates showing a slight advantage in favor of the three complex predictors over *gshare.fast*. At a hardware budget of 64KB, the perceptron predictor achieves a 3.6% misprediction rate, compared with 4.4% for a 64KB *gshare.fast* predictor. Thus, the complex predictors are more accurate than *gshare.fast*. Figure 6 shows, for each benchmark, the misprediction rates of the two complex predictors at a 64KB budget, as well as the misprediction rate of *gshare.fast* at a 64KB budget.

Multi-Component			**Perceptron**	***2Bc-gskew***
Hardware Budget	Delay (cycles)	Hardware Budget	Delay (cycles)	Delay (cycles)
18KB	3	16KB	4	3
30KB	3	32KB	4	3
53KB	4	64KB	5	4
98KB	5	128KB	7	5
188KB	7	256KB	8	7
368KB	9	512KB	11	9

Table 2. Predictor access latencies

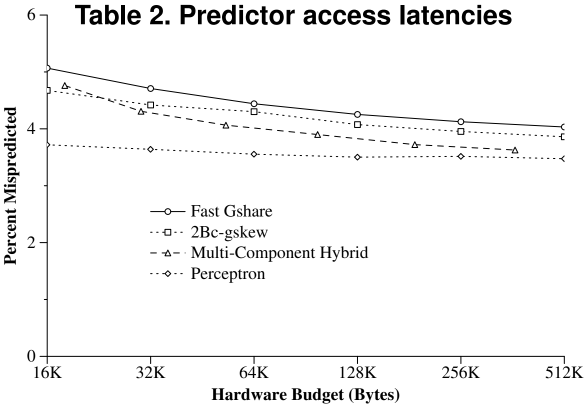

Figure 5. Arithmetic mean misprediction rates

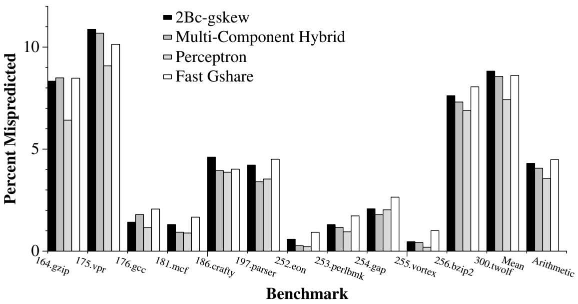

Figure 6. Misprediction rates at a 53KB budget

4.3 Ideal IPC

Figure 7 shows the harmonic mean IPCs produced by our cycle-accurate simulations. On the left, we show a graph with the ideal IPCs for each predictor, disregarding any impact of branch predictor latency. At a hardware budget of 53KB, the multi-component predictor achieves an IPC of 1.80, which is 4% higher than the IPC for *gshare.fast*

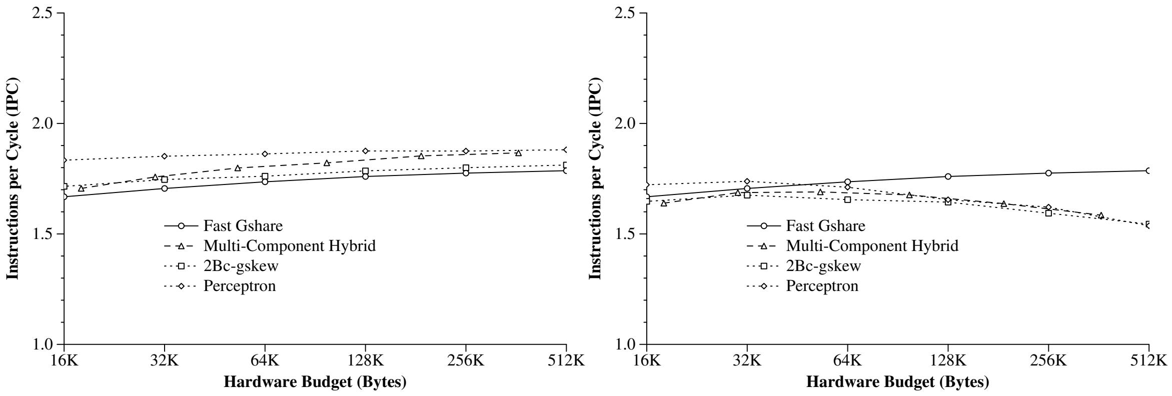

Figure 7. Harmonic mean IPC for 1-cycle prediction (left) and overriding prediction (right)

at 64KB, which is 1.73. The 64KB perceptron predictor achieves an IPC of 1.86, which is 7.5% higher than the IPC for *gshare.fast*. The two complex predictors would clearly allow better performance than our predictor given an ideal implementation, but the magnitude of the advantage is somewhat unimpressive.

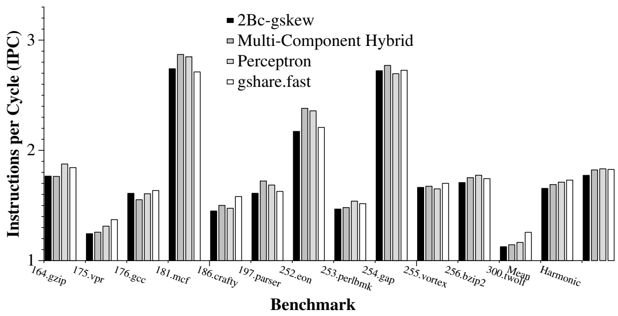

Figure 8. IPCs for each benchmark at 53KB hardware budget.

4.4 Realistic IPC

When branch predictor latency is taken into account, as in the graph on the right in Figure 7, the advantage of the two complex predictors vanishes. This graph shows the more realistic IPCs achieved when the two complex predictors are implemented as overriding predictors with a single-cycle 2K-entry *gshare* initial predictor. The IPCs for *gshare.fast* do not change, since that predictor has uses its special properties to operate in a single cycle. Because the overriding mechanism must sometimes be invoked, the small performance advantage of the complex predictors over *gshare.fast* turns into a slight loss. Figure 8 shows the IPCs for each SPECint 2000 benchmark, as well as the harmonic and arithmetic means, at at a hardware budget of 53KB. The perceptron predictor produces a harmonic mean IPC of 1.71, slightly lower than that of *gshare.fast*, which is still 1.73. The multi-component predictor achieves an IPC of 1.69, which is lower than that of *gshare.fast*. For some benchmarks such as `181.mcf` and `254.gap`, the complex predictors result in slightly higher IPCs than *gshare.fast*, but for others such as `186.crafty` and `300.twolf`, *gshare.fast* yields slightly higher IPCs. At larger hardware budgets with higher access delays, the complex predictors yield lower performance, despite their better accuracies, due to the higher overriding penalty. However, the point of presenting these figures is not necessarily that our branch predictor is strictly superior to the other predictors, but that the IPCs are about the same without the added overheads imposed by the complex predictors.

4.5 Explaining the Difference

For complex predictors there is a significant difference between the ideal IPC assuming single-cycle prediction and the realistic IPC when we take delay into account. The reason is that the overriding mechanism relies on a relatively inaccurate first level branch predictor. Each time the first and second level predictors disagree, a single cycle overriding penalty is incurred. The more accurate the second level predictor is, the more often the cost of overriding will have to be paid. perceptron predictor overrides the first level predictor an average of 7.38% of the time. On the benchmark `300.twolf` with the multi-component predictor, the first and second level predictors disagree 18.1% of the time.

5 Conclusions and Future Work

In future processor generations, the availability of large hardware budgets will favor conceptually simple branch predictors over complex predictors with high implementation overheads. This assertion is driven by two ideas. First, the overhead of implementing a complex predictor can eliminate the performance advantage it might have had because of its improved accuracy. Second, complex predictor organizations affect other aspects of the microarchitecture, increasing overall design complexity. We have presented a simple predictor that solves these problems with its low implementation overhead and isolation from the rest of the microarchitecture. We are currently studying ways to reorganize other predictors to take advantage of the same ideas. We conclude that future research into complex branch predictors must consider the impact of the extra complexity on performance.

6 Acknowledgments

I thank Calvin Lin and Samuel Z. Guyer for their helpful comments on an earlier draft, and I thank Emery D. Berger for inspiring the title of this paper.

References

[1] V. Agarwal, M.S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *the 27th Annual International Symposium on Computer Architecture*, pages 248–259, May 2000.

[2] Doug Burger and Todd M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[3] Hans de Vries. AMD's hammer microarchitecture preview. *Chip Architect*, October 2001.

[4] Karel Driesen and Urs Hölze. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings of the 31th International Symposium on Microarchitecture*, December 1998.

[5] Marius Evers. *Improving Branch Prediction by Understanding Branch Behavior*. PhD thesis, University of Michigan, Department of Computer Science and Engineering, 2000.

[6] M.S. Hrishikesh, Norman P. Jouppi, Keith I. Farkas, Doug Burger, Stephen W. Keckler, and Premkishore Shivakumar. The optimal useful logic depth per pipeline stage is approximately 6 fo4. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[7] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, pages 67–76, December 2000.

[8] Daniel A. Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4), November 2002.

[9] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[10] Scott McFarling. Combining branch predictors. Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

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

[12] Glenn Reinman and Norm Jouppi. Extensions to cacti, 1999. Unpublished document.

[13] Intel Corporation Press Release. Intel builds world's first one square micron sram cell. *Intel Press Room, http://www.intel.com/ pressroom/ archive/ releases/ 20020312tech.htm*, March 2002.

[14] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakakis Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[15] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, October 1996.

[16] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Computer Corporation, August 2001.

[17] Kevin Skadron, M. Martonosi, and D.W. Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction-Level Parallelism*, 2, January 2000.

[18] Li C. Tsai. A 1GHz PA-RISC processor. In *Proceedings of the 2001 International Solid State Circuits Conference (ISSCC)*, February 2001.

[19] Fred Weber. AMD's next generation microprocessor architecture. In *Microprocessor Forum*, October 2001.

[20] T.-Y. Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24^{th} ACM/IEEE Int'l Symposium on Microarchitecture*, pages 51–61, November 1991.

[21] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proceedings of the 7th ACM Conference on Supercomputing*, pages 67–76, July 1993.