Appears in the 2009 IEEE International Symposium on Performance Analysis of Systems and Software

Analysis of the TRIPS Prototype Block Predictor

Nitya Ranganathan Doug Burger† Stephen W. Keckler

Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu

† Microsoft Research
One Microsoft Way, Redmond, WA 98052
dburger@microsoft.com

Abstract

This paper analyzes the performance of the TRIPS prototype chip's block predictor. The prototype is the first implementation of the block-atomic TRIPS architecture, wherein the unit of execution is a TRIPS hyperblock. The TRIPS prototype predictor uses a two-step prediction process: it first predicts the exit from the current hyperblock and uses the predicted exit in conjunction with the current hyperblock's address to predict the next hyperblock.

SPECint2000 and SPECfp2000 benchmarks record average misprediction rates of 11.5% and 4.3%, respectively, on the prototype chip. Simulation-driven analysis identifies short history lengths, inadequate offset bits in the branch target buffers, and aliasing in the exit and target predictors as the main reasons for the predictor inefficiency. If the above issues are addressed, block misprediction rates can be reduced by 15% for SPECint2000 and 22% for SPECfp2000. Using a perceptron-based analysis, we show that there is significant loss in correlation in our current hyperblocks. We conclude that while carefully tuned block predictors can achieve relatively lower misprediction rates, new predictor designs and correlation-aware hyperblock formation are necessary to bridge the gap between block prediction accuracies and branch prediction accuracies.

1 Introduction

The TRIPS architecture [19, 20] is a distributed, tile-based design that evolved in response to increasing wire delays seen in modern process technologies. The distributed architecture tolerates wire delays through a combination of block-based, block-atomic execution [5, 16], and compiler-directed instruction placement and operand communication. TRIPS is the first architecture that uses an EDGE (Explicit Dataflow Graph Execution) ISA.

The unit of execution in TRIPS is the TRIPS hyperblock (termed as "block" in the rest of the paper). A hyperblock is a single-entry multiple-exit block of possibly predicated instructions [14]. Several heuristics aid the block construction process [13, 27]. Dynamically, the processor fetches and maps successive blocks on to the execution substrate using control-flow speculation, and executes the instructions within each block in a data-flow fashion. The block-based execution provides high fetch/execute/commit bandwidth and also amortizes the overheads of speculation, renaming, register reads, and completion.

Control-flow speculation is necessary for good performance in the TRIPS processor. The instruction window in the TRIPS prototype can hold up to eight blocks or 1024 instructions; hence, the cost of recovering from a misspeculation is high. Each block may have up to 128 useful instructions. A control-flow misspeculation can cause several such blocks to be flushed from the pipeline, resulting in a heavy re-fill penalty. The next-block predictor is at the heart of the control-flow speculation logic in the TRIPS architecture. A TRIPS hyperblock can have several exit branches (i.e. control transfer instructions that steer execution out of the current block). The predictor first predicts the exit that will fire and then predicts the target of this exit, which is the predicted next block address. Since the block predictor need only make predictions once per hyperblock, the number of predictions it must provide is far fewer than traditional branch predictors. However, the cost of misspeculation is high and hence, a high premium is placed on the accuracy of each prediction.

This paper performs a retrospective analysis of the TRIPS prototype chip's block predictor. While exhaustive design-space exploration studies [18] guided the design of the prototype next-block predictor, area constraints forced the designers to reduce the area allocated for the predictor. We report block misprediction rates from the TRIPS prototype chip and analyze the reasons for the large number of block mispredictions. We describe design improvements that help achieve 15% reduction in mispredictions for SPECint and 22% reduction for SPECfp benchmarks. Finally, we present a perceptron-based analysis to show the need for correlation-aware block formation and highlight the importance of local predictors. Our results show that using sophisticated exit prediction techniques and carefully tuned target predictors, we can achieve higher accuracies compared to the prototype predictor. However, new exit predictor designs and correlation-aware hyperblock formation are required to match state-of-the-art branch prediction accuracies.

2 Background

TRIPS [19, 20] is a block-atomic EDGE architecture that uses large single-entry multiple-exit blocks of possibly-

predicated instructions. Predication is used for control-flow within a block while branches are used to transfer control out of the block. A taken branch in the block leads to transfer of control to another block and is termed as an exit branch, or simply an exit. A hyperblock can have several exits but exactly one of the exits fires at run time. Every exit branch in the hyperblock is given a 3-bit identifier that is stored in the branch instruction. An exit predictor predicts the ID of the exit that will fire. By predicting an exit number *N*, the predictor, in effect, predicts a predicated path through the block which leads to exit *N*.

The TRIPS prototype (die photo shown in Figure 1) is a 130nm ASIC design with 170 million transistors and a maximum operating frequency of 366MHz. It is a tile-based design with two 16-wide issue processor cores and 1 MB of L2 NUCA (Non-Uniform Cache Access [11]) cache. Each of the processor cores contains five major tiles: Global Control Tile (GT, with predictor), Execution Tile (ET), Data Tile (DT), Instruction Tile (IT), and Register Tile (RT).

Implementing a block predictor instead of a traditional branch predictor has two advantages: more time to make a prediction (predictions made every eight or more cycles) and less complex implementation of speculative update support (histories checkpointed every block, not every branch). Because of these advantages, the prototype predictor uses a simple blocking 3-cycle design with single-ported structures and implements speculative updates for all histories.

However, a block predictor also has disadvantages compared to a branch predictor. In a distributed architecture like TRIPS, blocks of instructions fetched from the instruction cache banks are dispatched directly to the execution tiles, where they are decoded and executed. The GT manages the entire control for a block, namely, predict, fetch, execute, flush, complete, and retire. The predictor cannot examine the branches among the instructions dispatched to the ETs. Snooping the control network quickly for branch instruction opcodes and other fields is problematic as the GT is physically separate from the ETs to which instructions are dispatched directly from the caches. Hence, when a block is fetched, the prototype predictor knows only the block address and the block size.

3 Next-Block Prediction in TRIPS

In an early version of the TRIPS architecture, a simple 2-level block predictor similar to a 2-level global branch predictor was used [16]. Subsequently, a systematic design-space exploration [18] of block predictors was performed and the best-performing tournament exit predictors were determined. The Trimaran compiler [30] was used to generate hyperblocks that were mapped on to the TRIPS execution substrate [16, 19]. The goal was to arrive at a reasonably simple design for the exit predictor while providing large fetch bandwidth and high exit prediction accuracy. Our design space exploration considered local, global, and path histo-

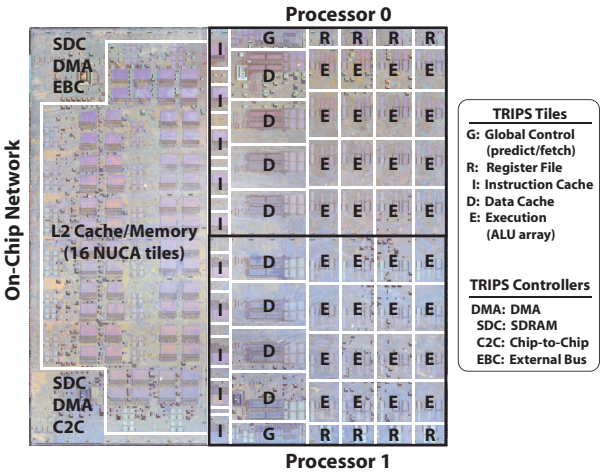


Figure 1: TRIPS prototype chip die photo with two 16-wide cores and 1 MB of L2 NUCA cache.

ries [17]. While a global history captures global branch/exit correlation information, a path history captures the address of the branches/blocks along the path leading to the current branch/block. The local/global tournament predictor was one of the best performing simple designs. The proposed design was similar in structure to the Alpha 21264 branch predictor [10]. More details and comparison with multiple branch prediction approaches can be found in [18].

The design decisions for the TRIPS prototype predictor were made based on results from the Trimaran hyperblock-TRIPS simulator infrastructure. However, for the TRIPS prototype, the software infrastructure was changed from the Trimaran compiler to the Scale compiler [27] targeting TRIPS. A new Instruction Set Architecture (ISA), timing constraints, and area constraints influenced the predictor design decisions. The TRIPS prototype predictor includes a scaled-down version of the local-global tournament exit predictor described in [18]. It also includes a multi-component target predictor for predicting various types of branches. Based on the GT area constraints, we implemented a 5KB (approx.) version of the tournament exit predictor and a 5KB (approx.) multi-component target predictor. The prototype exit and target predictor components with marked sizes are shown in Figure 2. The predictor configuration is shown in Table 1.

3.1 Prototype exit predictor

A TRIPS hyperblock contains at least one exit. Though a block may have several exits, only eight IDs are allowed (three bits in a branch instruction). The exit predictor needs to predict a 3-bit ID. The prototype exit predictor is a tournament predictor with 38Kbits (approx. 5KB) of total storage containing three components: a local exit predictor, a global exit predictor, and a choice predictor.

Local exit predictor: The local predictor has a 512-entry level-1 local (per-block) history table with each entry containing a 10-bit per-block local exit history (recent five exits represented using the last two bits of each exit). It also

Predictor and size	Predictor configuration
proto10K exit, 5KB	10-bit *lhist*, 512 local-l1, 1K local-l2; 12-bit *ghist*, 4K global-l2; 12-bit *ghist* for chooser, 4K choice-l2, 3-bit choice counter
proto32K exit, 16KB	12-bit *lhist*, 512 local-l1, 4K local-l2; 14-bit *ghist*, 16K global-l2; 14-bit *ghist* for chooser, 16K choice-l2, 3-bit choice counter
TAGE exit, 5KB	5-comp TAGE with history lengths 0 (tagless bimodal), 7, 27, 63, 112 & table entries 1K, 128, 512, 512, 1K
TAGE exit, 16KB	4-comp TAGE with history lengths 0 (tagless bimodal), 10, 55, 108 & table entries 4K, 1K, 2K, 4K
Local/TAGE exit, 5KB	8-bit *lhist*, 256 local-l1, 512 local-l2; 5-comp TAGE with history lengths 0 (tagless bimodal), 9, 18, 63, 108 & 512-entry tables; 2K choice-l2
Local/TAGE exit, 16KB	11-bit *lhist*, 512 local-l1, 2K local-l2; 3-comp TAGE with history lengths 0 (tagless bimodal), 20, 72 & table entries 4K, 1K, 4K; 8K choice-l2
5KB target, 5KB	4K-entry Btype predictor, 2K-entry BTB, 9-bit branch offset, 128-entry CTB, 9-bit return address offset, 128-entry RAS
5KB impr. target, 5KB	4K-entry Btype predictor, 2K-entry BTB, 11-bit branch offset, 128-entry CTB, 10-bit return address offset, 64-entry RAS
32KB target, 16KB	16K-entry Btype predictor, 4K-entry BTB, 9-bit branch offset, 512-entry CTB, 9-bit return address offset, 128-entry RAS
32KB impr. target, 16KB	16K-entry Btype predictor, 4K-entry BTB, 13-bit branch offset, 512-entry CTB, 10-bit return address offset, 64-entry RAS

Table 1: Predictor configurations of various exit and target predictors evaluated. *lhist* is local exit history and *ghist* is global exit history.

has a 1024-entry level-2 exit prediction table with each entry containing a 3-bit exit ID and a hysteresis bit used for replacement. The level-1 history table is indexed using the lower-order block address bits while the level-2 table is indexed using an XOR of the block address bits and the local history retrieved from the level-1 table [15].

Global exit predictor: The global predictor uses global exit history (12 bits containing last four exits represented using the full 3-bit IDs) to index into the level-2 global prediction table (4096 entries, each 4-bits wide). The global predictor indexes the level-2 table using an XOR of the block address and the history.

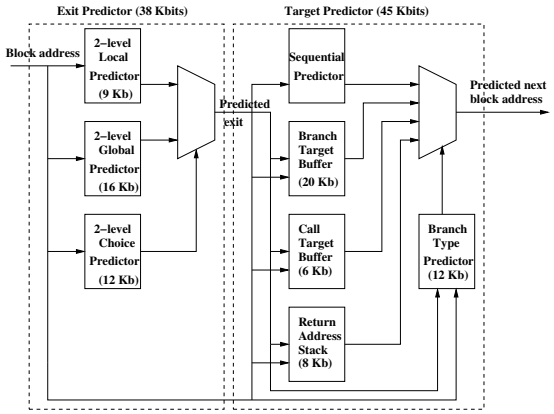


Figure 2: Prototype block predictor components.

Choice predictor: The choice or chooser predictor is similar to the chooser found in the Alpha 21264 processor [10]. It uses global histories (12 bits containing the last six 2-bit truncated exits) and a second level table (4096 entries, each 3-bits wide) of saturating counters. The indexing is similar to the global predictor indexing. The counters predict global or local depending on the most significant bit.

Speculative updates and recovery: The prototype predictor implements speculative updates of the local, global, and choice histories. Since the instruction window can hold only a maximum of eight blocks, the speculative state that has to be maintained is much lesser than in conventional instruction-atomic superscalar architectures. For the local predictor, an 8-entry CAM structure is used to maintain the latest histories while the global and choice history registers are backed-up in 8-entry history files [26]. On a misspeculation leading to a pipeline flush, the correct histories are restored accurately.

3.2 Prototype Target Predictor

The prototype target predictor predicts the next block address using the current block address and the predicted exit from the exit predictor. Since exit branches can be of many types, support to predict various types of targets like branch target, call target, and return target is necessary. Typically, modern target predictors have multiple components, each tuned to predicting the target for one type of branch. The prototype target predictor has four components for predicting four types of branches, and a branch type predictor. We describe each component below:

Branch type predictor (Btype): Due to the prototype's distributed design, the block predictor does not have access to the branch instruction fields at prediction time. This limitation necessitates predicting the branch type for the predicted exit. The type of the branch can be learned by the predictor at completion time when the branch resolves. The 4096-entry Btype predictor predicts one of four types: sequential branch, branch, call, and return. The sequential branch type is learned internally to prevent sequential exits ("fall-through" branches whose target is the the next hyperblock in program order) from occupying BTB entries. Each Btype entry stores a 2-bit type and a 1-bit hysteresis.

Sequential branch target predictor: In block-based architectures, the sequential exit branch target has to be computed, unlike in conventional processors where the program counter is automatically incremented to point to the next sequential instruction after every fetch. Using an adder logic, the sequential target is computed from the current block address and the current block size.

Branch Target Buffer (BTB): The direct-mapped tagless BTB has 2048 entries with each entry containing a 9-bit signed offset and a 1-bit hysteresis. The branch target is computed by adding the shifted offset to the current block address. Since the branch instruction cannot be read at prediction time to retrieve the offset for direct branches, or all the targets computed and stored in the header due to space constraints [6], a BTB is necessary to store offsets and predict branch targets. As indirect branches are not separately indicated by the Btype predictor, the BTB is used for predicting indirect branches also.

Call Target Buffer (CTB): The call target is stored as an absolute address since call targets can be located far away from the call instructions. Every entry of the 128-entry CTB has a call target and a return address offset (offset from the

calling block) corresponding to the return associated with the function that is called. Every entry contains two hysteresis bits, one each associated with the call and return targets.

Call-Return Mechanism and Return Prediction: The call-return mechanism is different in TRIPS compared to conventional instruction-atomic architectures. To ensure block atomicity, function return points must be at the start of a block because entering a block in the middle (after returning from the call) violates block-atomic constraints. Since a block may contain many calls and each call may have a different return address (not necessarily the address of the next block), storing all the return addresses in the block header is not feasible. Hence, return addresses are learned dynamically using a link stack that learns call-return relationships at commit time. The learned return addresses are stored in the CTB. The Return Address Stack (RAS) is used to predict returns. The 128-entry RAS is pushed on a predicted call and popped on a predicted return at fetch time.

Misprediction recovery: All target prediction structures except the RAS are updated only at commit time. When a block is fetched, the top of stack pointer and value are checkpointed and used later when recovery is initiated. This simple technique provides almost-perfect recovery [26].

SMT mode: In the TRIPS SMT (Simultaneous Multithreading) mode, up to four threads can execute concurrently. The predictor design includes several optimizations for prediction in the SMT mode: sharing of prediction and target buffers using the thread address space ID, maintaining separate global histories for all threads and partitioning the RAS into four 32-entry stacks.

Predictor area and timing: The predictor occupies 70% of the Global Tile area of $2mm^2$. The predictor has a simple non-pipelined blocking design. It takes three cycles for prediction and three cycles for update. Exit prediction and target prediction are initiated in parallel. The exit prediction is complete at the beginning of the third cycle, and the final target selection is complete at the end of the third cycle.

3.3 Predictor Evaluation

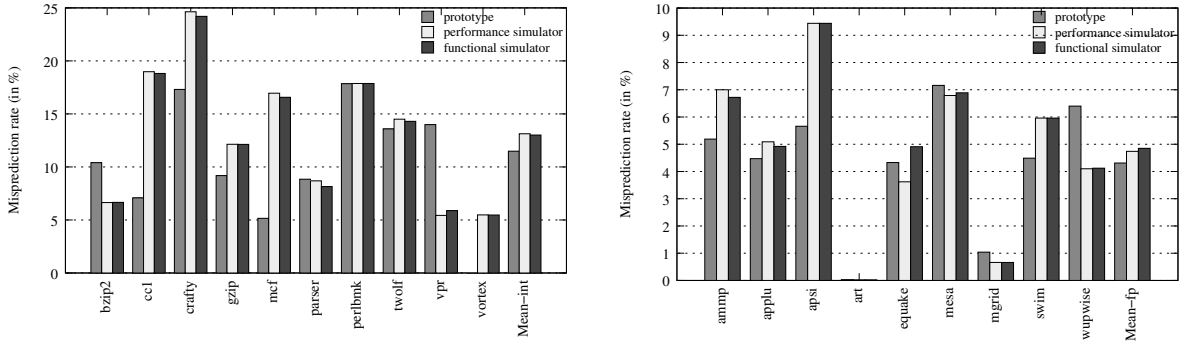
The Scale compiler infrastructure [27] was used to generate hyperblocks for the TRIPS prototype. We use the highest level of Scale compiler optimization (-Omax) along with function inlining and loop unrolling. We use the SPEC2000 benchmark suite for evaluation (ten SPECint and nine SPECfp benchmarks, the full set of benchmarks that we were able to compile successfully). Two metrics for predictor evaluation are used: Misprediction Rate and MPKI (Mispredictions per Kilo Instructions).

Figure 3 shows the misprediction rates of SPECint and SPECfp benchmarks. The first bar for each benchmark shows the misprediction rate from the prototype, the second bar shows the misprediction rate from the TRIPS cycle-accurate performance simulator and the third bar shows the misprediction rate from the functional branch predictor simulator. All benchmarks were run to completion (with reference inputs) on the prototype. Since *vortex* did not complete successfully on the prototype, we do not show prototype results for *vortex*. For the performance and functional simulators, we show results for Simpoint [24] regions (equivalent to approximately 100M RISC instructions). The misprediction rates for SPECint are 11.49%, 13.13%, and 13.01% respectively and the rates for SPECfp are 4.31%, 4.74%, and 4.85% respectively, for the prototype, performance, and functional simulators. The mean misprediction rates for the block predictor are higher than conventional branch predictors (which achieve integer misprediction rates of 5% or less). The Simpoint regions provide a good approximation of the entire benchmark suite for our evaluation based on misprediction rates except for few benchmarks like *gcc*, *crafty*, and *apsi*. Furthermore, the difference between cycle-accurate performance simulation and functional simulation is negligible, because speculative updates are implemented for histories and the RAS which mimics the immediate update of predictor state in the functional model. In the rest of this paper, we report results exclusively from the functional simulator.

4 Analysis of block prediction

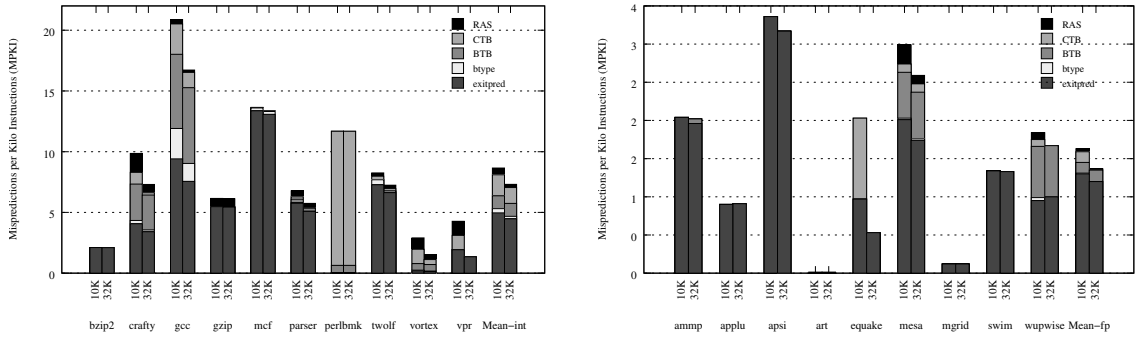
In this section, we characterize block mispredictions in the TRIPS prototype predictor, show bottlenecks in the predictor design, and suggest design modifications to improve the predictor performance. We examine history types, history lengths, table types, exit predictor types, and target component types to arrive at an improved block predictor. We use the 10KB TRIPS prototype predictor (*proto10K*) as well as a scaled-up 32KB version of the prototype predictor (*proto32K*) for the experiments. Since the 10KB predictor is severely size-constrained for some component optimizations, we also include the 32KB predictor, which is a reasonable size for future processors.

Figure 4 shows the component-wise misprediction breakdown (MPKI) for *proto10K* and *proto32K*. The lower-most component in each bar represents the MPKI due to the exit predictor. The next component shows the MPKI from the branch type predictor when the exit predictor has predicted correctly (since the branch type prediction depends on the exit predictor). Similarly, the next three components show the BTB, CTB, and RAS prediction components and their contribution to mispredictions when the exit and the branch type predictions are correct (since target prediction depends on exit and branch type). For some benchmarks the total value of the component MPKIs is slightly greater than the MPKI reported in the previous section. Occasionally, the target predictor hides the exit predictor inefficiencies i.e., the predicted target is correct even when the predicted exit is wrong. The graph shows that for most of the benchmarks, the major contributor to MPKI is the exit predictor; the exit MPKI is more than 50% of the total MPKI for both SPECint and SPECfp programs. The exit predictor MPKI is 4.96 for *proto10K* and 4.48 for *proto32K*. For some benchmarks like



(a) SPECint 2000 benchmarks (b) SPECfp 2000 benchmarks

Figure 3: Prototype predictor misprediction rates for SPECint and SPECfp benchmarks.



(a) SPECint 2000 (b) SPECfp 2000

Figure 4: MPKI breakdown of prototype predictor (10KB) and scaled-up prototype predictor (32KB) by component type.

gcc, *perlbmk*, and *equake*, other components like the BTB and the CTB perform poorly. There is a reduction in both the exit and the target MPKIs when the larger 32KB predictor is used. The overall MPKI for SPECint benchmarks is 7.76 and 6.49 respectively, for the 10KB and the 32KB configurations. For SPECfp benchmarks, *proto10K* achieves an MPKI of 1.63 while *proto32K* achieves an MPKI of 1.37. On the whole, the exit predictor, the BTB and the CTB are the major contributors to block predictor mispredictions.

4.1 Analysis of exit prediction

We now analyze exit predictor mispredictions. Figure 5 shows the breakdown of mean exit MPKI by "misprediction reason" for *proto10K* (containing a 5KB predictor) and *proto32K* (containing a 16KB predictor). For SPECint benchmarks, aliasing in the local, global, and choice predictor level-2 tables results in 18.1% (0.9 MPKI) of the exit mispredictions for *proto10K* while there is less aliasing in the *proto32K* predictor (12.5%, 0.56 MPKI). An imperfect chooser results in 26.8% (1.33 MPKI) of the exit mispredictions in the prototype predictor while it contributes 29.7% (1.33 MPKI) of the mispredictions in the scaled-up 32KB predictor. The remaining mispredictions (contributing more than half of the total exit mispredictions) are due to predictor design drawbacks such as insufficient history length and simple prediction algorithm (2-component tournament predictor instead of a multi-component predictor), and block construction issues such as placement and predication of correlated branches (discussed in Section 5). For the SPECfp suite, aliasing is negligible, especially in the larger predictor. Chooser inefficiencies contribute to about one-third of the total exit mispredictions, while the majority of the mispredictions (about two-thirds) is due to prediction technique and predication issues.

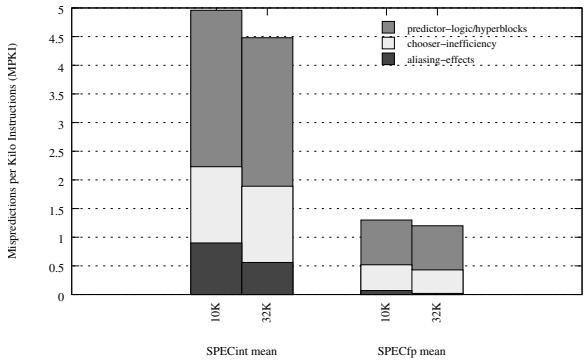


Figure 5: Mean exit MPKI breakdown for SPECint and SPECfp benchmarks for the 10KB prototype (5KB exit predictor) and the 32KB scaled-up prototype (16KB exit predictor) predictors.

To understand the reasons for exit prediction misses due to predictor design issues, we evaluated several types of history-based exit predictors (address-indexed or bimodal, local, global, path) and combinations of history-based predictors. Due to space constraints, detailed quantitative results for our scaling, aliasing, history-length, and

component-type variation experiments are not presented. We summarize the relevant results below and describe one experiment motivating the need for multi-component long-history predictors.

Single and multi-component exit predictors using different prediction component types

In general, for exit prediction using a 1-component predictor, we found that global exit histories are slightly better than path histories and path histories are better than local exit histories. Global and path history-based predictors scale well (up to 128KB) when the history length and prediction table (level-2) size are increased. When considering 1-component and 2-component predictors of similar size, the local/global tournament predictor and the bimodal/path tournament predictors are the best-performing predictors for almost all the sizes from 1KB to 256KB. The number of bits of each exit to use in the exit history is the same as for the prototype for a local/global tournament predictor. Our experiments showed that the prototype exit predictor configuration gives the best tournament predictor among such 5KB exit predictors.

Multi-component exit predictors using global and path history-based components

Recent branch prediction research has shown the potential of using more than two components with short and long history lengths to make a branch prediction [12, 21]. Branches may be correlated with different branches from the past, near-branches or far-branches. Some branches are predictable with short histories while some are predictable with long histories [3, 28]. We explore multi-component predictors consisting of global history-based components and multi-component predictors consisting of path history-based components. The components can use nine history lengths ranging from zero to 30 bits. If exit IDs are truncated to two bits, a 30-bit global exit history can hold 15 exits which can represent anywhere from 15 to 120 branches depending on the number of exits in each block. However, several branches that could be represented in a traditional branch history will not be represented in the exit history due to predication and exit truncation. For path histories, we use a few lower-order bits of the address of each block encountered in the execution path leading to the current block.

To evaluate the potential of multi-component exit predictors, we simulate interference-free predictor components to remove the aliasing benefits of large-history predictors compared to small-history predictors, while retaining only the history length benefit for our analysis. We also avoid folding the histories. In a practical implementation, we may use history folding, smaller tables that can lead to destructive aliasing and fewer component tables to reduce the predictor area. Finally, we consider an ideal chooser which always chooses the component that predicts correctly, provided some component predicts the correct exit ID. We use integer benchmarks for our evaluation since they are significantly more difficult to predict than FP benchmarks. Also, integer benchmarks have more branches that show variations in correlation.

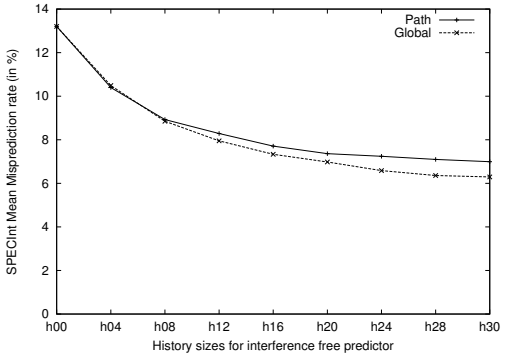


Figure 6: Mean misprediction rates for global and path interference-free predictor components using history lengths from 0 to 30.

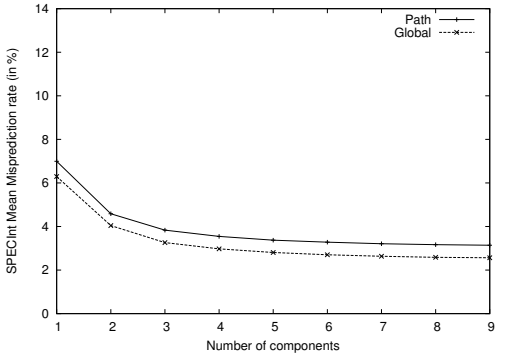


Figure 7: Mean misprediction rates for the best *N*-component predictor for *N* ranging from 1 to 9. Both global and path-based multi-component predictors are shown.

We evaluate all possible multi-component predictors containing one to nine components. Figure 6 shows the mean misprediction rates for each of the predictor components using history lengths from 0 to 30 bits. There are two mean curves, one for global components and one for path components. This graph shows how well each of the individual components perform. The misprediction rate drops steadily from 13.2% for a 0-bit history to 6.3% for 30-bit global history and from 13.2% for a 0-bit history to 7.0% for 30-bit path history.

Figure 7 shows the mean misprediction rates achieved by the best *N*-component predictor for *N* ranging from 1 to 9. For a given *N* value, a point on the X-axis corresponds to the set of all N-component predictors, and the corresponding Y-axis value gives the misprediction rate for the best N-component predictor in that family of predictors. For both global and path predictors, the misprediction rates go down rapidly as more components are included in the hybrid predictor. Even with just two components, the predictor is significantly better than the individual predictors shown in Figure 6. The best 2-component global predictor has a misprediction rate of 4% while a 2-component path predictor

achieves a rate of 4.6%. Predictors with six or more components perform equally well with a misprediction rate of 2.7% for a 6-component global predictor and a misprediction rate of 2.6% for a 9-component global predictor. Since we use interference-free predictors, the low misprediction rates seen in these hybrid predictors is mainly due to the combination of short and long history lengths used. When more histories are used, each block has a higher chance of being predicted using the history length that is the "best" for predicting the block effectively. On average, the global predictor performs better than the path-based predictor for all history lengths.

Table 2 shows the history lengths of the global component predictors for the best performing configuration for each length *N* from Figure 7. It also shows the corresponding lowest misprediction rate. The components for each of the best configurations use histories from either end of the set of histories as well as from the middle of the histories, if they can use more. We also observe that more histories are chosen closer to each other at the beginning but when moving towards the longest history, fewer such histories are chosen. This pattern of history selection indicates an approximate geometric trend [21] in the choice of history lengths. About 85% of the branches can be predicted well by the 0-bit history component. The number of branches that require long histories is few; hence, fewer components using the longer histories are chosen. For example, in a global 5-component predictor, three components (0 bit, 4 bit and 12 bit) are from the lower half of the set of histories and two components (20 bit and 30 bit) are from the upper half. This experiment shows us the potential of multi-component predictors using increasing history lengths.

4.2 Analysis of target prediction

We now examine target predictor inefficiencies in the prototype predictor. Figure 4 shows that, for SPECint, the target predictor's contribution to mispredictions goes down when the target predictor is scaled from 5KB to 16KB. For SPECfp benchmarks, the low BTB and CTB MPKI in *proto10K* become close to zero in *proto32K*. To determine the limits of target prediction accuracy, we simulated multi-component target predictors with a perfect exit predictor. The main bottlenecks in the prototype target predictor (with perfect exit prediction) for SPECint are described below:

- A 4096-entry Btype predictor was insufficient to store the types of all exit branches. The Btype MPKI goes down by half in the *proto32K* predictor (0.51 to 0.27) when the number of entries is increased 4-fold.
- The maximum branch offset length for direct branches within the same function was determined by considering the maximum number of blocks in large functions. The BTB offset was set to nine bits. After the prototype implementation was complete, we evaluated our predictor using hyperblocks from the Scale compiler. Our results showed that several functions in the SPECint suite had more hyperblocks than we had estimated. Thus, we had underestimated the branch offset width.
- Aliasing was high in the BTB and CTB tables. In general, 128 entries in the CTB were sufficient, but some benchmarks with many small functions performed poorly. When scaling the prototype predictor to 32KB, little reduction in the BTB MPKI and a significant reduction (26%) in the CTB MPKI were observed.
- Though the RAS size was halved in *proto32K*, the RAS MPKI goes down because the CTB size in *proto32K* is 4X the size of the CTB in *proto10K* leading to less aliasing for return address storage in the CTB entries.
- Absence of a separate indirect branch predictor was significant for benchmarks like *crafty*, *gcc* and *perlbmk*.
- Few RAS mispredictions are caused by RAS overflow and underflow.

The BTB and the CTB have the maximum contribution towards target mispredictions. If offset length mispredictions are removed, the mispredictions from these components can be partitioned into aliasing and indirect branch/call target mispredictions. We examine taken exits in the dynamic execution stream to identify the number of unique dynamic targets for each taken indirect branch. Some indirect branches are easily predictable by the BTB, e.g. those that have only one target during the entire run. If there is more than one dynamic target, we categorize the branch/call as a dynamic indirect branch/call. The MPKI is calculated separately for such indirect branches and indirect calls. The remaining mispredictions are due to aliasing in the target buffers. For some benchmarks, indirect branches significantly contribute to the MPKI. For example, *gcc* has several multi-target branches that are difficult to predict while *perlbmk* has several multi-target calls that are difficult to predict. These branches need a separate indirect branch predictor to achieve good prediction accuracies. On average, we found that for the 16KB target predictor (*proto32K*), about 75% of the mispredictions from the BTB and 84% of the mispredictions from the CTB are indirect branch/call mispredictions.

4.3 Improving exit and target prediction

Inspired by our analysis of ideal multi-component predictors showing the potential of predictors with multiple histories (mostly, geometrically increasing lengths), we evaluated block predictors inspired from two recent branch predictors, the O-GEHL predictor [21] and the TAGE predictor [23]. The O-GEHL exit predictor did not outperform the prototype predictor (due to an inefficient final chooser and table aliasing). Hence, we present the TAGE predictor below.

TAGE exit predictor: The TAGE branch predictor proposed by Seznec et al. [23] makes use of a set of geometrically increasing history lengths indexing a set of tagged tables. This branch predictor can be directly mapped to exit

Num. Components	1	2	3	4	5	6	7	8	9
Best global mispred. rate (%)	6.29	4.04	3.26	2.97	2.81	2.70	2.63	2.59	2.57
Global history lengths chosen	30	0, 30	0, 12, 30	0, 8, 16, 30	0, 4, 12, 20, 30	0, 4, 8, 16, 24, 30	0, 4, 8, 12, 16, 24, 30	0, 4, 8, 12, 16, 20, 24, 30	0, 4, 8, 12, 16, 20, 24, 28, 30

Table 2: Best global multi-component predictor configurations for different numbers of components from one to nine.

prediction since the final prediction is chosen based on tag match and not adder-tree computation (adder-tree is more suitable for binary prediction). A TAGE predictor consisting of five components is shown in Figure 8.

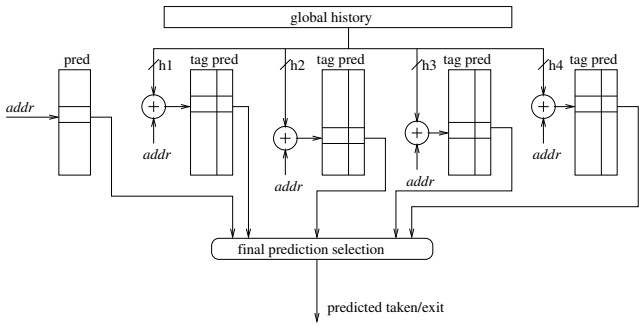


Figure 8: A 5-component TAGE predictor containing a bimodal component and four global+path history indexed components with geometrically increasing history lengths (*0*, *h1*, *h2*, *h3*, *h4*).

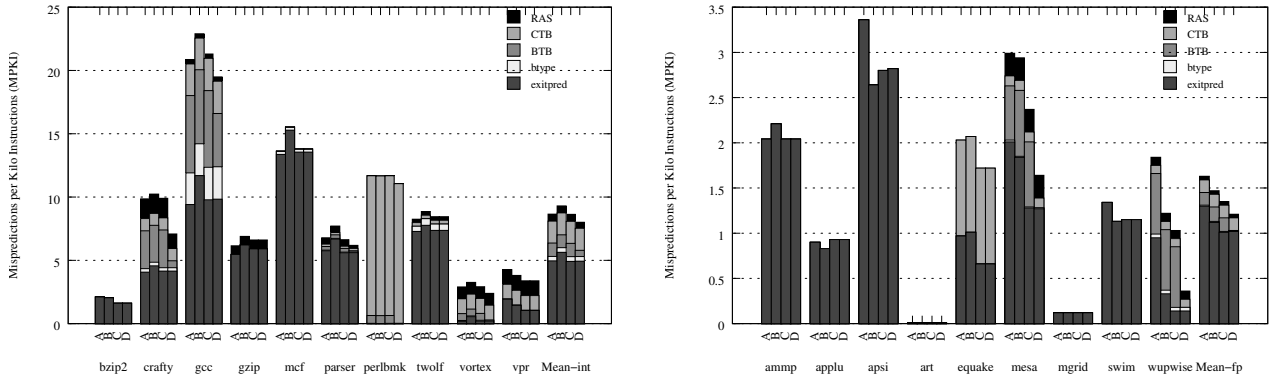
The TAGE predictor is inspired from the ideal PPM predictor [1] that uses all history lengths from 0 to N-1 for an N-component prediction. By combining the geometric history length technique of O-GEHL with the ideal PPM approach, an implementable version of the PPM predictor, i.e., TAGE is obtained. The index to each of the tables is generated using a hash of the corresponding history length, path history bits and branch address. The prediction tables are tagged to help ensure that a component giving a correct prediction for a branch is the only component making a prediction for that branch. In a practical design, often, two components are updated. Hence, prediction for a branch occurring in a particular path is never made by more than two tables. This design saves space in the other components for other branches and reduces replication of predictions. The disadvantage is that tags occupy lot of space and a fine balance is required between the number of table entries and tag length. When a branch can be predicted well by a component using a short history length, it is not allocated to longer-history components. This optimization reduces destructive aliasing and helps the longer-history components predict the difficult branches better. Typically, the first component is a tag-less bimodal predictor. When a branch needs to be predicted, the component for which the tag matches gives the final prediction. If more than one component has a matching tag, the longer-history component is chosen. When none of the components have matching tags, the bimodal component makes the prediction. TAGE can also use dynamically varying histories and replacement thresholds.

Adapting a TAGE branch predictor to predict exits is straightforward. The TAGE exit predictor's final prediction selection logic is similar to that of the TAGE branch predictor. There are two main differences in our TAGE implementation. First, we do not use dynamic history or threshold fitting. Second, we use a hash of the path history and the block address bits as the tag in each TAGE table instead of the block address alone. Using the history also to calculate the tag achieves lower MPKI. We evaluated different sizes of TAGE and local/TAGE hybrid predictors with varying number of tables, history lengths, tag lengths and hysteresis bit lengths. The best TAGE and local/TAGE predictor configurations are shown in Table 1.

Target predictor: For target prediction, we focus on reducing the BTB and the CTB MPKI as they are the biggest sources of target mispredictions. The BTB design space exploration showed that small offset width, aliasing and indirect branch mispredictions are the main sources of inaccuracy. The 9-bit offset used in *proto10K* and *proto32K* was not sufficient to include the targets of several blocks in certain benchmarks. The benchmark that is most affected by the low offset width is *gcc*. Increasing the offset length to 13 bits (from nine bits), removed nearly all the mispredictions due to insufficient offset length. For the 5KB predictor, we increase the BTB offset by two bits.

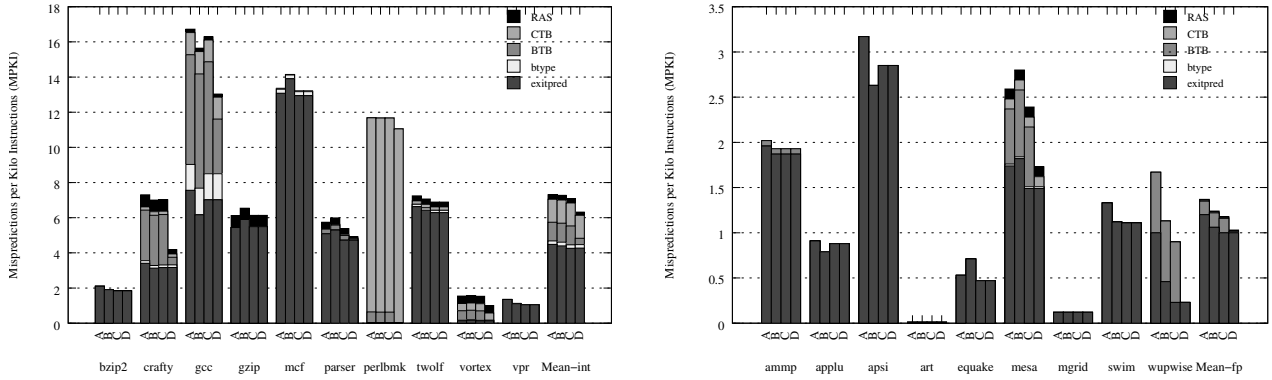
The CTB MPKI accounts for CTB aliasing and indirect call mispredictions. The CTB stores full call target addresses and return address offsets. Hence, offset width mispredictions are possible for returns only. Wrong return addresses being pushed on to the RAS cause return address mispredictions. When the CTB size is increased and the return offset length in the CTB is increased by one bit, the RAS MPKI goes down. The remaining mispredictions can be addressed by using associative target buffers to reduce aliasing and by including a separate indirect branch predictor. We do not explore these aspects in this paper. Table 1 shows the improved target predictor configurations.

Predictor evaluation: Figure 9 shows the MPKI breakdown of several predictors for integer and FP benchmarks. The first bar shows the MPKI breakdown for the *proto10K* predictor as seen earlier. The second bar shows the MPKI of a block predictor with the TAGE exit predictor and a target predictor identical to the *proto10K* predictor. The third bar shows the MPKI of a block predictor with the local/TAGE hybrid exit predictor and a target predictor identical to the



(a) SPECint 2000 (b) SPECfp 2000

Figure 9: Comparison of MPKI breakdown for 10KB predictors for SPECint and SPECfp suites. A indicates *proto10K*, B is TAGE with the same target predictor as *proto10K*, C is Local/TAGE with the same target predictor as *proto10K* and D is Local/TAGE with the improved target predictor.



(a) SPECint 2000 (b) SPECfp 2000

Figure 10: Comparison of MPKI breakdown for 32KB predictors for SPECint and SPECfp suites. A indicates *proto32K*, B is TAGE with the same target predictor as *proto32K*, C is Local/TAGE with the same target predictor as *proto32K* and D is Local/TAGE with the improved target predictor.

proto10K predictor. The final bar shows the MPKI of a block predictor with the local/TAGE hybrid exit predictor and the improved target predictor. Figure 10 shows the MPKI breakdown for the 32KB configuration. We allowed a 10% variation in predictor size to optimize different components.

For integer benchmarks, the TAGE predictor has a higher exit MPKI than the *proto10K* predictor while the local/TAGE predictor performs as well as *proto10K* (exit MPKI of 4.92). In a 32KB configuration, the TAGE predictor performs slightly better than *proto32K* while the local/TAGE predictor has an MPKI that is 5% lower (exit MPKI of 4.26). We also implemented a large 64KB local/TAGE exit predictor. This predictor has 12.5% lower MPKI than a 64KB local/global tournament predictor. Using the improved target predictor with the local/TAGE predictor achieves the lowest overall MPKI. On the whole, the overall MPKI drops from 7.76 to 7.15 from *proto10K* to the local/TAGE predictor with an improved target predictor. This represents an improvement of 7.9%. For the 32KB configuration, the improvement is higher: the overall MPKI goes down by 15% (from 6.49 to 5.42). For SPECfp benchmarks the mean bars show that there is a steady improvement in the MPKI from left to right for both the sizes The overall MPKI values for the four configurations are 1.56, 1.42, 1.34 and 1.19 for the 10KB predictors and 1.3, 1.22, 1.17 and 1.01 for the 32KB predictors. These numbers indicate an improvement of 23.7% for the 10KB configuration and 22.3% for the 32KB configuration. For both sets of benchmarks the best exit MPKI is achieved by the local/TAGE predictor for most of the benchmarks (exceptions are *crafty*, *gzip* and *twolf* among SPECint and *applu*, *apsi* and *swim* among SPECfp). The TAGE branch predictor is a sophisticated predictor that achieves very low MPKI. However, the TAGE exit predictor only performs on par with the local/global tournament predictor. When combined with the local predictor, it performs much better. These numbers indicate the potential of using improved predictors using multiple length histories along with local exit predictors in future block predictor designs.

5 Branch correlation in hyperblocks

In the previous section, we evaluated several exit predictors. The best exit predictor that used long global histories, tagged tables and a sophisticated algorithm was able to eliminate only an additional 5% of the exit mispredictions. We hypothesize that loss in correlation is the reason for the poor performance of exit predictors when compared to state-of-the-art branch predictors [8, 23].

One goal of hyperblock formation is to enhance pre-

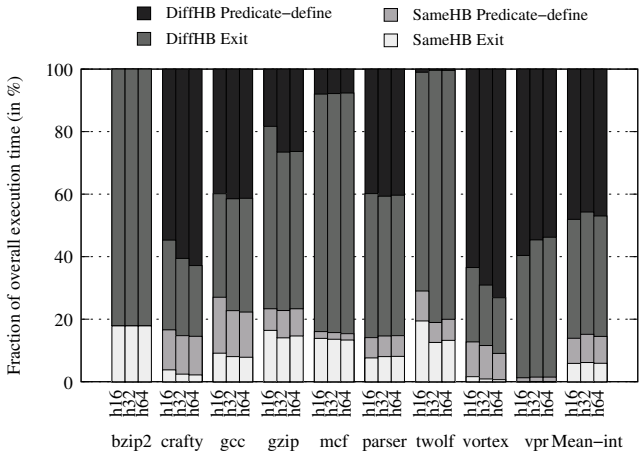


Figure 11: Dynamic distribution of correlated branches among hyperblocks as exits and predicate-defines. Bars marked h16, h32 and h64 indicate correlation observed within global branch histories of length 16, 32 and 64 bits respectively.

dictability by hiding difficult-to-predict branches as predicates. Improving the predictability of blocks alone does not guarantee increased performance. For example, basic blocks are more predictable than hyperblocks using good branch predictors but basic blocks are small and offer insufficient scope for aggressive compiler optimizations. Furthermore, when executing basic blocks, the window is severely underutilized in a block-based processor like TRIPS, which leads to low performance. Aggressive block formation by combining too many basic blocks to construct large hyperblocks can convert correlated branches to predicates and create more exit branches. Both of these effects result in poor predictability. Hence, a balance must be struck between the number of useful instructions in the block and block predictability. To that end, we perform a perceptron-based branch correlation analysis to analyze whether hyperblock formation masks branch correlation and reduces the efficacy of global history-based predictors.

5.1 Comparing exit and branch predictability

Predicting exits can be more difficult than predicting branches because of possible loss in correlation among exits. Also, predicting exits is a "1 of N" problem (where $N \leq 8$ for the TRIPS prototype) while predicting branches is a "1 of 2" problem. We evaluate misprediction rates and MPKI for -O3 (basic block, BB) and -Omax (hyperblock, HB) compiled codes from the Scale compiler. We describe our findings for the 16KB exit predictor in *proto32K* for SPECint benchmarks. SPECfp benchmarks show a similar but less pronounced trend. The *proto10K* exit predictor also shows the same trend. To predict basic blocks, we ensure that a branch predictor with the same size and the same local/global tournament structure is used.

We observed that the mean misprediction rate for the BB binary is 40% lower compared to the rate for the HB binary (4.79% vs 7.97%). Lower misprediction rates indicate that the branch predictor is inherently better at making predictions compared to the exit predictor. The MPKI values indicate an opposite trend. The mean MPKI for hyperblock code is 4.48 while it is 7.68 for basic blocks. This is because, even though the exit predictor is predicting worse than the branch predictor, it has to make many fewer predictions which leads to a large reduction in the MPKI. Since MPKI is the real indicator of the impact of branch mispredictions on performance, we see that the hyperblock code is, in effect, better performing than the basic block code. This justifies the use of hyperblocks instead of basic blocks for improved performance. If the predictability of exits can be improved to equal or exceed the predictability of basic blocks, we can achieve further reduction in the MPKI, leading to larger performance improvements.

5.2 Understanding exit predictability using correlation analysis

To identify the loss of correlation due to block construction mechanisms, we perform a dynamic correlation analysis of each branch in the BB code and identify all the other branches that strongly influence the direction of this branch. A global interference-free perceptron predictor [9] is used to understand the set of correlated branches (in near and far histories) for each BB code branch. Our compiler tags each branch with a unique ID in the BB phase of compilation and carries over the tag in all the subsequent phases as well. Thus, using these tags in the intermediate code from the HB phase, we can determine how the BB branches are mapped to the HB exits and predicate-defines.

During a run of the HB code, we determine for each taken block exit, whether the branches with which it is correlated occurred in the same hyperblock or in a different hyperblock, and whether the correlated branches occur as exits or predicate-defines. Figure 11 shows the weighted distribution of placement of strongly correlated branches in hyperblocks. We consider strongly correlated branches as those that have 75% or more correlation with a given branch. Perceptrons with 16, 32, and 64-bit global branch histories were used to understand how much correlation information hyperblock codes stand to lose vis-a-vis basic blocks running on hardware with various global history lengths (16, 32, 64). We show nine of the SPECint benchmarks in this study (*perlbmk* did not work with our infrastructure).

A strongly influencing branch can be in one of the following four places: within the same hyperblock as an exit branch (bottommost component in each bar), within the same hyperblock as a predicate defining instruction (second component), within a different block as a branch (third component) and within a different block as a predicate-define instruction (topmost component). If the correlated branch is in the same block as the current exit branch, the global predictor may not be able to use the direction information of the correlated

branch if it did not occur in the recent history captured by the global predictor. If the correlated branch is in the same block or in a different block, but has been converted to a predicate-define, we once again lose correlation information that was present in the BB compiled code. Simon et al. have proposed incorporating predicate information in global branch history [25].

There can be significant loss of correlation in the HB code due to the presence of correlated branches as predicates and exit branches in the same hyperblock. Dynamic correlated branches tend to occur within the same hyperblock as much as 15% of the time as exits (9%) or as predicate-defines (6%). Some of the correlation provided by exits within the same hyperblock may be captured by a global predictor if the recent history in the predictor contains multiple instances of the current block. However, in general, a local predictor can capture self-correlation (in this case within the same block) more easily. About 47% of the dynamic correlated branches are found as predicates in another hyperblock. Considering predicate-defines within the same block and within different blocks, more than half of the strongly correlated branches are present as predicate-defines in the hyperblock code leading to a significant loss in correlation. Since the remaining correlated branches comprise of branches present as exits in different hyperblocks, they can be captured by global predictors when they occur in recent history.

These results indicate that there is severe loss in correlation among exits in hyperblocks. Sometimes, a branch may be strongly correlated with several other branches and even if some of them are predicated, it could still be predicted accurately if the directions of the non-predicated correlated branches are available. For future work, we plan to analyze the influence of correlated branches by classifying branches that are necessary for the current branch to be predicted correctly and branches that may be if-converted without loss in accuracy. "Lost" correlation from predicated branches can be reclaimed by using predicate values (both predicted and resolved) in the exit predictor. The set of correlated branches for each branch can be profiled and used in the compiler to construct better hyperblocks.

6 Related Work

Previous studies have proposed multiple branch and block prediction to enable high fetch bandwidth [2, 22, 31]. These predictors were used to predict few basic blocks at a time. Block prediction and exit prediction for block-based processors have been explored in [5, 6, 7]. In our framework, direct target prediction methods as proposed by some of these previous studies have not been as effective as exit prediction followed by target prediction. Exit prediction is also less complex and more accurate than previous multiple branch prediction proposals [18]. To our knowledge, block-atomic call-return semantics and assumptions about branches for distributed processors (e.g. number of exits in a block, type of branches) have not been discussed by previous studies.

Correlation, branch behavior and limits of branch predictability have been explored in detail for global predictors [1, 4]. We are not aware of a similar analysis for exits and predicate-defines in hyperblocks. Prior to this work, Loh used an interference-free perceptron to study the global correlation amongst branches in regular programs to motivate partitioned history predictors [12]. Our use of perceptrons is to understand the global correlation loss that arises due to correlation-agnostic hyperblock construction heuristics. Thomas et al. [29] used dataflow-based value tracking across instructions to find the set of branches in the recent execution window affecting a branch's direction. Using this, they removed non-correlated branches from the global history for future predictions. Several researchers have examined the interaction of predication and branch predictability. Simon et al. discuss the notion of *misprediction migration*, where only one of a pair of correlated branches get predicated, thereby leaving the second branch without valuable correlation information in the dynamic history [25].

7 Conclusions

This paper presented the design, evaluation, and analysis of the TRIPS prototype block predictor. The TRIPS prototype predictor has several features like support for branch type prediction, block-atomic call-return semantics, speculative updates for all histories, and SMT mode. Misprediction rate measurements from the prototype were deconstructed with detailed software simulations to understand the component-wise breakdown of the predictor MPKI. Design-space exploration experiments show that a local/TAGE exit predictor performs 5% better than a local/global exit predictor but still falls short of accuracies achieved by advanced branch predictors like the TAGE branch predictor. For target predictors, we observed that changing the offset widths in the BTB and the CTB results in the elimination of a significant number of target mispredictions. On the whole, compared to the 10KB prototype predictor, an improved block predictor containing a local/TAGE exit predictor and an improved target predictor achieve 7.9% and 23.7% reduction in MPKI for SPECint and SPECfp benchmarks respectively. For the scaled-up 32KB predictor, the improvements are better: 15% and 22.3% for SPECint and SPECfp respectively. Using a perceptron-based correlation analysis, we highlighted the need to make block construction heuristics aware of information about correlated branches, so that correlation-aware placement and predication of branches in hyperblock code can be implemented in the compiler. This experiment also served to reaffirm the importance of a good local exit predictor component in predicting hyperblock exits, since intra-block correlation is captured well by the local component.

In summary, we have shown that, with moderate hardware, block prediction can be made feasible and reasonably accurate which will enable higher execution efficiency in

block-based architectures. We showed design changes that achieve reasonable reduction in mispredictions. However, to lower the MPKI further, designing better exit predictors and employing indirect branch predictors may be necessary. New prediction algorithms utilizing all types of available intra-block and inter-block correlation as well as correlation-aware block formation heuristics may make block predictors perform on par with branch predictors.

Acknowledgments

This research is supported by a Defense Advanced Research Projects Agency contract F33615-01-C-4106 and by NSF CISE Research Infrastructure grant EIA-0303609.

References

[1] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, 1996.

[2] S. Dutta and M. Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *International Symposium on Microarchitecture*, pages 258–263, Dec. 1995.

[3] M. Evers, P.-Y. Chang, and Y. N. Patt. Using hybrid predictors to improve branch prediction accuracy in the presence of context switches. In *International Symposium on Computer Architecture*, pages 3–11, Jul 1996.

[4] 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 *International Symposium on Computer Architecture*, pages 52–61, July 1998.

[5] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *International symposium on Microarchitecture*, pages 191–200, December 1996.

[6] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control flow speculation in multiscalar processors. In *International Symposium on High Performance Computer Architecture*, pages 218–229, Feb. 1997.

[7] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *International Symposium on Microarchitecture*, pages 14–23, Dec. 1997.

[8] D. Jiménez. Piecewise linear branch prediction. In *International Symposium on Computer Architecture*, pages 382–393, Jun 2005.

[9] D. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *International Symposium on High Performance Computer Architecture*, pages 197–206, Jan. 2001.

[10] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[11] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, Oct 2002.

[12] G. H. Loh. A simple divide-and-conquer approach for neural class branch prediction. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 243–254, Sep 2005.

[13] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Head and tail duplication for convergent hyperblock formation. In *International Symposium on Microarchitecture*, pages 65–76, Dec 2006.

[14] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Microarchitecture*, pages 45–54, 1992.

[15] S. McFarling. Combining branch predictors. Technical Report TN-36, DEC WRL, Jun 1993.

[16] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *International Symposium on Microarchitecture*, pages 40–51, Dec 2001.

[17] R. Nair. Dynamic path-based branch correlation. In *International Symposium on Microarchitecture*, pages 15–23, 1995.

[18] N. Ranganathan, R. Nagarajan, D. Jiménez, D. Burger, S. W. Keckler, and C. Lin. Combining hyperblocks and exit prediction to increase front-end bandwidth and performance. Technical Report TR-02-41, Department of Computer Sciences, The University of Texas at Austin, Sep 2002.

[19] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *International Symposium on Computer Architecture*, pages 422–433, Jun 2003.

[20] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethmadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *International Symposium on Microarchitecture*, pages 480–491, Dec 2006.

[21] A. Seznec. Analysis of the O-GEmetric History Length branch predictor. In *International Symposium on Computer Architecture*, pages 394–405, Jun 2005.

[22] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. In *Architectural Support for Programming Languages and Operating Systems*, pages 116–127, 1996.

[23] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length predictor. *Journal of Instruction-level Parallelism*, Feb 2006.

[24] T. Sherwood, E. Perelman, and B. Calder. Basic block istribution analysis to find periodic behavior and simulation points in applications. In *Internationl Symposium on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.

[25] B. Simon, B. Calder, and J. Ferrante. Incorporating predicate information into branch predictors. In *International Symposium on High Performance Computer Architecture*, pages 53–64, Feb 2003.

[26] K. Skadron, M. Martonosi, and D. Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction-level Parallelism*, 2, Jan. 2000.

[27] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, March 2006.

[28] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 170–179, Oct 1998.

[29] R. Thomas, M. Franklin, C. Wilkerson, and J. Stark. Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In *International Symposium on Computer Architecture*, pages 314–323, Jun 2003.

[30] Trimaran: An infrastructure for research in instruction-level parallelism. http://www.trimaran.org.

[31] T.-Y. Yeh, D. Marr, and Y. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *International Conference on Supercomputing*, pages 67–76, Jul 1993.