Incorporating Predicate Information Into Branch Predictors

Beth Simon Brad Calder Jeanne Ferrante

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{esimon,calder,ferrante}@cs.ucsd.edu

1. INTRODUCTION

The Explicitly Parallel Instruction Computing (EPIC) architecture has been put forth as a viable architecture for achieving the *instruction level parallelism* (ILP) needed to keep increasing future processor performance [8]. The IA-64 Itanium processor [1] is an example of an EPIC architecture. An EPIC architecture issues wide instructions, similar to a VLIW architecture, where each instruction contains many operations.

One of the new features of the EPIC architecture is support for *predicated execution* [14], where each operation is guarded by one of the predicate registers available in the architecture. An operation is committed only if the value of its guarding predicate is true.

One advantage of predicated execution comes from predication's ability to combine several smaller basic blocks into one larger region. This provides a larger pool from which to draw instruction level parallelism (ILP) for EPIC architectures. Another advantage of predicated execution is that it can eliminate hard-to-predict branches by translating them into predicate defines, which do not need to be predicted. This comes at the cost of executing both paths following the branch as if it were a single path.

Choi et al. [7] recently performed a study, where they reported that only 7% of cycles are spent due to branch mispredictions for the SPEC 2000 integer benchmarks (without using if-conversion). This is partially due to the in-order Itanium processor stalling because of memory latencies, which end up shadowing the stalls due to branch mispredictions. As this type of EPIC architecture progresses and memory latencies are better hidden, the stalls due to branch mispredictions will have a much larger impact.

The goal of our research is to use predicated execution to see how low we can make the branch misprediction rate by removing all of the the hard-to-predict branches, not caring about the increase in executed code created via predication. We first examine how to create predicated regions to remove hard-to-predict branches. In order to aggressively form predicated regions around these hard-to-predict branches, we had to leave unbiased, but originally predictable, branches (conditionals, unconditionals, and returns) inside predicated regions. We call branches left inside predicated regions *region branches*.

The creation of predicated sequences (region formation) based on removing a hard-to-predict branch can have a negative impact on the predictability of these region branches. Region branch execution is now predicated on a register defined by a predicate compare definition that was added in order to remove the hard-to-predict branch. These region branches will need to be predicted during fetch more frequently than they were in the original, non-predicated code (i.e. a region branch will be fetched both when its guarding predicate will be true and when it will be false). This can cause what we call *misprediction migration*, where the poorly predictable pattern of a hard-to-predict branch that was eliminated due to predication is merely migrated to a region branch. In addition, direct branches (e.g., unconditionals and returns) that are left in the region are also affected by misprediction migration. Before the region was formed, these region branches were accurately predictable as taken. After region formation, they now need to be predicted as either taken or not-taken when guarding predicate is TRUE, and should always be predicated as not-taken when the guarding predicate is FALSE. Because of misprediction migration, we found little improvement in branch misprediction rate for some programs when using a traditional branch predictor for region formation targeted at hard-to-predict branches.

In this paper, we examine two new branch predictor optimizations. First is a new branch prediction optimization called *Squash False Path* (Squash-FP) that attempts to know a branch's guarding predicate value as it is being fetched, and, if it is false, then the branch is predicted as not-taken. The goal of this predictor is to correctly predict the region branches that are on the false path as not-taken.

The second predictor we examine adds predicate information into the global history register. We examine the *Predicate Global Update Branch Predictor* (PGU) architecture that incorporates predicate information into the global history to try and improve the performance of region-branches that benefit from correlation. The PGU predictor updates the global history with the predicate result when the predicate defining instruction is resolved. This can allow region branches to benefit from this history correlation when making their prediction from the global prediction table.

Region branches only benefit from these two branch prediction architectures if they are scheduled far enough apart from their predicate definitions. Therefore, we examine the benefit of rescheduling the predicated region to move the region-based branches as far away as possible from their predicate defining instructions. This attempts to increase the cases where a predicate define can be resolved before the branch is fetched, so it can be used to form the prediction for the branch.

2. PRIOR WORK

2.1 Branches and Region Formation

Previous region formation techniques have focused on using predicated execution to group basic blocks from various control flow paths into one region to improve compiler optimization opportunities and scheduling [12] [4] [3]. These hyperblocks are typically formed from an inner-most loop body. Basic blocks are incorporated into a region based on a heuristic function that weighs the block's frequency of execution and size in terms of instructions in relation to the main path of the hyperblock being formed. Hyperblocks target unbiased branches by translating them into predicate defines and incorporating the subsequent paths in the predicated region. Hyperblocks only allow heavily biased branches to remain as predicated branches in the region and incorporate the frequent path of execution as part of the region. Our region formation methods will target the removal of only unpredictable branches rather than unbiased branches.

2.2 Interaction Between Predication and Branch Prediction

Mahlke et al. [11] investigated the interaction of predicated hyperblock region formation and branch prediction using two branch prediction architectures: a BTB with a 2-bit counter, and a BTB with profile-based direction prediction. Over a subset of SPEC92 benchmarks and UNIX utilities, they showed a reduction in branch mispredict rate of 56% using hyperblock regions. Their work avoids the issue of having to predict predicated branches in regions by ensuring via hyperblock formation that only very infrequently taken branches are left in predicated regions.

Tyson [18] utilizes predicated execution to optimize short forward branches, showing that these constitute a significant percent of both integer and floating point branches and have relatively poor prediction rates. He shows up to a 30% reduction in misprediction rate for the SPEC92 benchmark suite – noting that most of the reduction comes directly from the branches translated into predicate defines that no longer require prediction. Tyson presents results for a region formation method that does not contain any changes in control flow. In addition, he examines an idealized region formation that allows any type of branch to remain in the predicated region, but states that it is unclear how to predict them. We assume for the results in [18], that branches left in these idealized regions are only being predicted when their guarding predicate is true.

2.3 Including Predicate Information in Branch Prediction

In [5], August et. al. presents a modified branch prediction architecture called the *Predicate Enhanced Prediction* (PEP) architecture that incorporates predicate information into a local per-branch prediction scheme. They elaborate on one of the problems of region formation by showing how the transformation of an unbiased branch into a predicate define could cause a previously predictable branch to become unpredictable. Their technique focuses on exploiting the relationship between a given predicate define and a branch guarded by that predicate to recover the original prediction pattern for that branch. This is accomplished by storing the guarding predicate register number of the branch instruction in the BTB. Additionally, two local histories are stored in the BTB entry – one associated with the branch behavior when the guarding predicate is true, and one when it is false. The theory is that "true history" should be used and updated with the same pattern that the original branch accessed – since it will be used if the branch's guarding predicate is true. The "false history" should be used when the branch's guarding predicate is false – hopefully producing a prediction of "not taken". However, realistically, whatever value is currently stored in the predicate register file when a branch is fetched is used to choose between histories. In cases where the predicate define guarding a branch has been issued to the pipeline, but not yet resolved, one may access the "false history" even when the branch's guarding predicate eventually resolves to true. Conversely, one may access the "true history" even when the branch' guarding predicate is false if a predicate value from some previous part of the code set that predicate true and the predicate define that produces a value for the current branch has not yet committed.

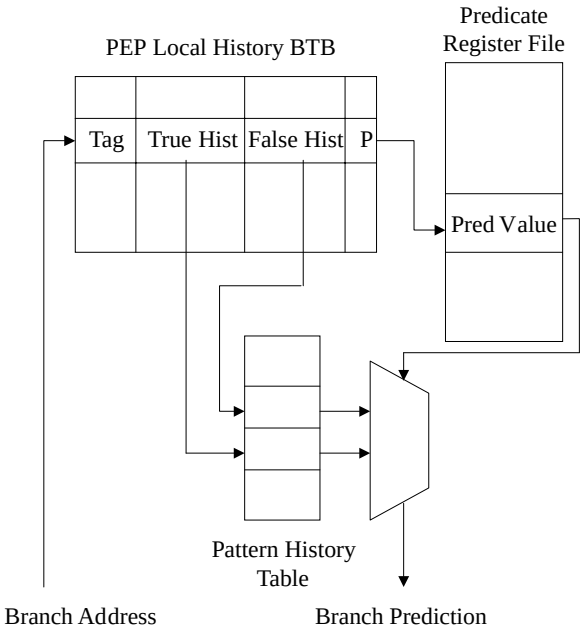


Figure 1: High level design of the Predicate Enhanced Prediction Architecture.

Figure 1 shows a schematic of the PEP branch prediction architecture. A branch prediction takes 2 serial table lookups. The first lookup accesses the BTB providing the guarding predicate associated with the branch. Then another lookup is made in the predicate register file to find the currently available predicate value. The predicate value is then used to choose between the predictions found by indexing a 2-bit pattern history table using the two histories from the BTB. We assume that the local histories are speculatively updated at fetch and correctly recovered on a branch misprediction.

Mahlke et. al [13] examined a new use of predicate registers for collecting information to assist in branch prediction via compiler synthesized information. They proposed new compiler techniques for statically examining register values to produce a dynamically executed function that would

help guide branch predictions. They use predicate registers to hold the result of this dynamically executed function, then this predicate value is used in a modified version of the prepare-to-branch instruction that precedes a branch in their architecture.

Klauser et al. [10] investigated predicated region formation for simple branch hammocks (`if-then` or `if-then-else` constructs only). They examined updating the global history register in the branch prediction hardware with predicate define information. They found only minor to no improvement from updating the global history register because they have no predicated branches in the regions that were formed. In comparison, our approach leaves unbiased branches in the predicated region, and these branches can benefit directly from having the predicate defines update the global history register.

3. PREDICATED REGION FORMATION

Since the cost of predication is directly tied to the cost of falsely guarded instructions executed in a region, intelligent region formation is paramount. At the same time, the benefit that can possibly be derived from predication is directly tied to the original negative impact of the hard-to-predict branches removed by predication. A clear starting point in evaluating the impact of predication is to form regions starting at the most frequently mispredicted branches, with the hope of greatly reducing the number of mispredictions. This would result in (1) not having to predict these very hard-to-predict branches and (2) the removal of frequent and "poorly behaving" entries from the branch prediction hardware, which can reduce destructive aliasing among the remaining branches. However, not all of these important hard-to-predict branches allow for region formation as simple as if-then-else-join conversion. For example, when analyzing branch mispredictions in the SPEC95 benchmark `go` we find several issues that complicate region formation. Most notably, there are several returns that are reached along frequent paths from the most hard-to-predict branches. We find the need to include these return statements in predicated regions if we are to affect any change in the branch misprediction rate of `go` via predication.

3.1 Our Region Formation Algorithm

Our region formation algorithm starts from a list of hard-to-predict branches that we target for translation to predicate define instructions. For the experiments presented here, we start from a list of the top 10% most frequently mispredicting static branches in each benchmark. Original mispredict values are gathered with a baseline Meta Chooser predictor [9] which is detailed in Section 4.

For a given hard-to-predict branch, we walk the control flow graph following the branch incorporating basic blocks into the predicated region. We continue adding successor blocks in a breadth-first fashion until we reach a depth of five basic blocks from the hard-to-predict branch. If, while walking, we encounter another member on the list of hard-to-predict branches, the depth count along that path is reset to zero. This method attempts to target the most frequently mispredicting branches for removal and provides sufficient quantity of post-branch work to overlap the execution of the branch converted to a predicate definition in the pipeline. Additionally, this method provided sufficient scope for our scheduler to investigate a range of region schedules, as will be discussed in Section 7.6.

If a block in the region originally ended in a branch and both of its control flow successor blocks have also been included in the region, then the branch is translated into a predicate define and the successor blocks are assigned the appropriate guarding predicates. If either of the block's successors were not included in the region, then the branch becomes a region branch.

There are additional measures we use in controlling region formation. First any successor block that is reached less than 10% of the time the region is entered is excluded from the region. This keeps cold blocks from unnecessarily bloating the region with infrequently useful work. Second, any block ending in an indirect branch or return automatically stops region formation along that path. Finally, any branch with a successor that has already been included in a previously formed region (such as `(c<a)`), stops region formation along that path.

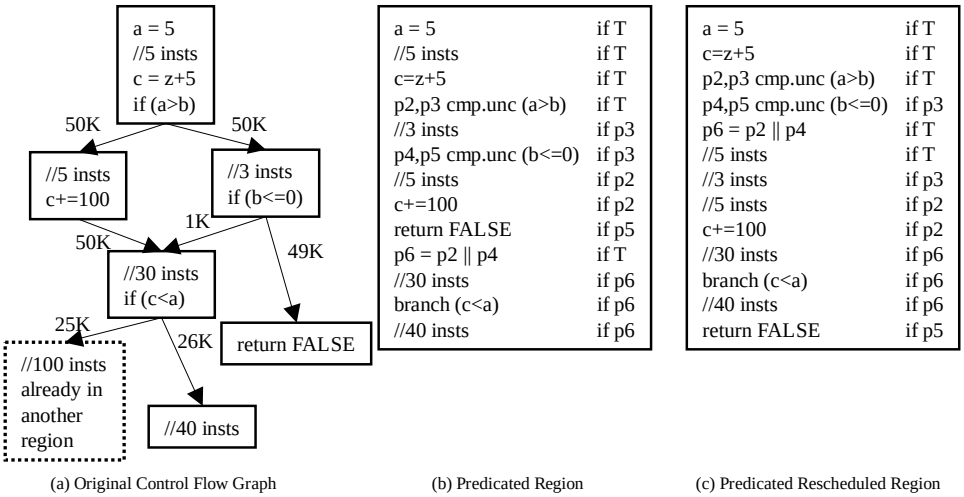
3.2 Issues with Region Branches

Consider the code in Figure 2(a). Assume that the branch `(a>b)` is a hard-to-predict branch we would like to replace with a predicate define. Applying our hard-to-predict region formation algorithm can produce the region in Figure 2(b). Statement `(b<=0)` is translated into a predicate define because its successors are reached more than 10% of the time we enter the region.

In our example, branch `(c<a)` (very predictable in the original code) is left as a region branch because its taken successor was already incorporated into a region. This leaves only the fall-through successor in the region. Though this is a highly unbiased branch it is very predictable in the original code because of its correlation with branch `(a>b)`. One of the most important aspects of this region is the presence of branches within it with one or more targets outside the region. These branches, like all instructions in the region, are tagged with guarding predicates. These branches will be fetched regardless of the value of their guarding predicate, and at commit, their effect (i.e. whether they are taken or fall-through) is contingent on both their condition (in the case of a conditional branch) and the value of their guarding predicate. In cases where the guarding predicate is false, regardless of branch condition, the branch is not taken.

As can be seen in this example, some always-taken branches (e.g., `return FALSE`) are transformed via the predication process into branches that must be predicted. Though they are still the same type of branch, whether or not they should actually be taken is contingent on whether their guarding predicate is true or false (i.e. whether their path is live or spurious). Hence, these branches now need to be predicted similarly to a conditional branch during fetch. In addition, conditional region branches are now dependent on a combination of conditions when they are predicted. While the branch condition still determines when the branch should be taken, if a conditional branch is predicated on false, it will be treated as a not-taken branch.

In general, all region branches will need to be predicted more frequently than they were in the original code. When we enter a region of predicated code we fetch all instructions down all paths, so we will be predicting branches some number of times more than their actual path of execution is being followed. In Figure 2(b) branch `return FALSE` will be fetched, predicted, and have to be predicted 100K times,



(a) Original Control Flow Graph (b) Predicated Region (c) Predicated Rescheduled Region

Figure 2: Region formation example that leaves unbiased, possibly predictable branches in the region. Solid boxed basic blocks are formed into a predicated region, the dotted basic block is not included in the region. In this work, we use abbreviated EPIC branch representation to show the condition evaluation as part of the branch. In the original code, branch (a>b) is our targeted hard-to-predict branch. Branch (b<=0) is highly biased and highly predictable. Branch (c<a) is not highly biased, but also originally very predictable given knowledge of the behavior of branch (a>b).

approximately twice as frequently as in the non-predicated code. This impacts the predictability of these instructions and can significantly increase the accesses to the branch prediction hardware.

All of these issues mean that, when using a traditional branch prediction architecture, the region branches in Figure 2(b) suffer from misprediction migration. This is because branch (c<a) and the return branch are both guarded by predicates defined by what was the unpredictable branch (a>b). Therefore, these branches ((a>b) and "return FALSE") are harder to predict using a traditional branch prediction architecture. Branches like the return branch should benefit from a local predicator utilizing predicate information as long as the define of predicate P5 can be scheduled sufficiently before the return. The branch (c>a) which may be predictable based on correlation with (a>b) should benefit from a global history scheme that can incorporate the information produced from the predicate definition of (c>a) even though neither P2 nor P3 is the guarding predicate of (c<a).

4. BASELINE BRANCH PREDICTOR

Our baseline branch predictor is a Meta Chooser [9] style predictor pictured in Figure 3. For the results in this paper, we simulated a 4K entry local, global, and chooser tables using a 12 bit global history register. This type of branch predictor takes advantage of both per-branch local history as well as recent path global branch history in making accurate predictions. This predictor uses the global table to make a prediction in a single cycle, which is squashed and updated if the local prediction made in the next cycle is selected by the chooser.

4.1 Baseline Meta Chooser Predictor

When using the baseline Meta Chooser predictor for predicated code, all region branches still speculatively update the speculative global history register during fetch. One difference in predicated regions is that all direct region branches (e.g., unconditional, returns, etc.) are now treated as branches that need to be predicted as taken or not-taken. Therefore, these branches update the global history register and obtain their direction prediction from the Meta Chooser predictor. For example, a predicated return branch instruction inside of a region determines that the next fetch PC would either be (1) the top of the return stack (taken), or (2) the fall-through PC (not-taken) based upon the direction prediction of the Meta Chooser. Conditional region branches use the Meta Chooser as they do in non-region code to produce a branch prediction. However, the Meta Chooser branch prediction really represents the *combination* of the guarding predicate and the evaluation of the conditional expression when predicting a branch.

In the baseline Meta Chooser predictor, all region branches speculatively update the speculative global history register and the local history register when they are fetched. The update the 2-bit state counters when the branch commits *even if guarded on a false predicate*. Falsely guarded branches are treated as if the branch evaluated to not-taken, and the branch state for a branch guarded on a false predicate will be updated as not-taken.

5. SQUASHING FALSE BRANCHES

The first predicate aware branch prediction architecture we propose uses guarding predicate knowledge to completely squash the prediction of falsely guarded branches. This is a modification of the use of predicate knowledge as defined by August et al. [5]. Their Predicate Enhanced Prediction (PEP) architecture uses the value of a branch's guarding predicate to choose one of two local history registers to use in predicting the branch – channeling branch predictions where the guarding predicate is known to be true to one 2-bit predictor, and those with false or yet-to-be defined guarding predicates to a different 2-bit predictor.

The *Squash False Path* (Squash-FP) architecture we pro-

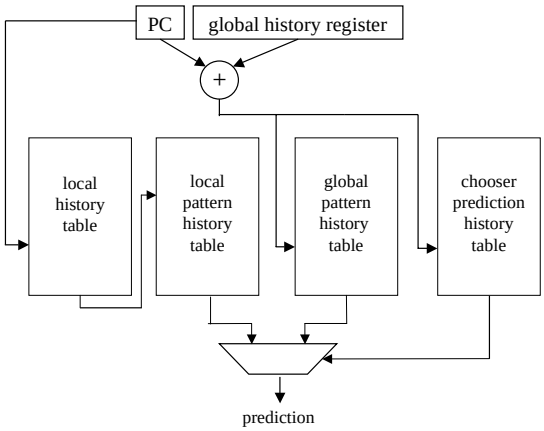


Figure 3: Our baseline Meta Chooser branch prediction architecture. It contains a local history predictor, global history predictor, and a 2-bit chooser table to choose between the two predictions.

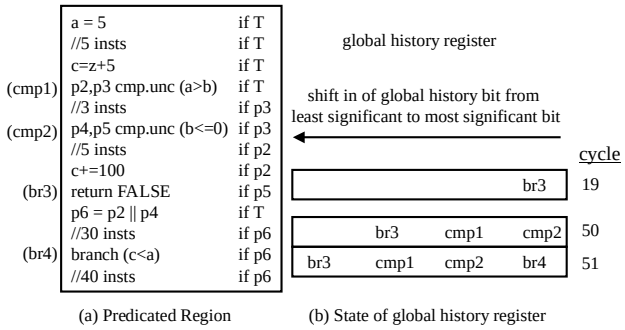


Figure 4: Update of the Meta Chooser global history register when enabled with predicate update. Note that branches update the global history register in fetch, while predicate defining instructions update it in writeback. This causes a "re-ordering" of information in the global history register as compared with instruction fetch ordering. For the purposes of this example, we show a fixed 20 instruction delay from predicate define fetch to resolve. In our results updates occur based upon the resolution latency for individual predicate define instructions.

pose stores the predicate register number that is guarding each branch in the branch's BTB entry. During a prediction the predicate register is looked up in the register table, and the lookup returns not only the value of the predicate, but also if the predicate has any outstanding definitions in the pipeline. A predicate register that has no outstanding definitions in the pipeline is said to be *resolved*. If the predicate is resolved and it evaluates to false, then we accurately predict not-taken for the branch. If the predicate evaluated to true or it was not resolved, then the default predictor is used. Not only will this give 100% prediction accuracy to those falsely guarded branches whose guarding predicates are resolved by the time they are fetched, but it will reduce contention in the tables for the remaining predictions.

To provide this prediction, we rely upon the register lookup to tell us if the latest definition has written to the register, or whether an instruction in the pipeline has yet to produce its value. This information is provided in traditional architectures to determine if a bypassed value from the pipeline should be used instead of the register file value when executing an instruction. We use this same information already provided by processors to tell if the predicate is resolved or not as described above.

6. GLOBAL UPDATE PREDICTOR

We will now examine incorporating predicate information into the global history based branch predictor. Our *Predicate Global Update* (PGU) predictor architecture stores the predicate result from predicate define statements in the speculative global history. When a predicate define instruction finishes its execution, it shifts the result of its condition evaluation into the lowest bit of the global history register in exactly the same way the result of a branch is shifted into the global history register. However, for predicate defines, this occurs when the predicate define value is known in the writeback stage. Since the predicate define instructions perform a *delayed update* of the speculative global history register, the ordering of the global history information will be different than in the original non-predicated program. A delayed (out-of-order) update of the global history register results in very accurate predictions as long as branch/predicate define ordering is similar throughout execution.

For a branch to benefit from predicate information stored in the history register, the predicate update has to occur before that branch is fetched and predicted. This can be addressed by compiler scheduling of the predicate defines as early as possible in the predicated region combined with scheduling the region branches as late as possible. This comes at the cost of executing additional non-true path instructions, since we exit the region later when a branch is taken out of the region.

Region branches (both spurious and true-path) update the PGU predictor in exactly the same fashion as the Baseline Predicated predictor. Both are predicted and speculatively update the global history register in fetch.

6.1 Speculative Global History Update

Figure 4 shows an example of how the predicate update predictor would update the global history register. For this example, we assume a 20 instruction delay from the fetch of a predicate define until it resolves in the writeback stage where we update the speculative global history register. The actual update would occur at varying times for different predicate define instructions depending on their latency from fetch to writeback.

Using the predicate update branch predictor, the speculative global history register is updated with the values of predicate registers `p2` (cmp1) and `p4` (cmp2). The predicate defines formed via our region formation process all define two complementary predicates, and we model the architecture such that it updates the speculative global history register with the value of the first predicate. The IA-64 architecture supports a variety of predicate define instructions that can define up to two predicates using many different boolean combinations [2]. Examining how to use other IA-64 predicate define instructions and their interaction with the predicate update predictor is an area for future work.

As can be seen in Figure 4 our predicate update predictor achieves its goal of incorporating the important history of

statement (a>b) into the speculative global history register. When (c<a) is fetched, the second most recent bit of history in the global history register (cmp1) helps determine the correct prediction for (c<a). This alleviates the problem of misprediction migration that would otherwise manifest for this branch without a predicate update branch predictor.

Using the schedule in Figure 4, the region branch return FALSE is not able to benefit from the predicate information in the global history, since global history from neither of the cmps is updated in time. A possible solution to this issue is to make predicate region code scheduling aware of the correlative behaviors between predicate defines and branches. A conservative solution to the problem is to schedule predicate defines as early as possible in a region while also scheduling branches as late as possible in the region. Figure 2(c) shows a rescheduling of the region in (b) where the number of intervening instructions between predicate define (c<a) and branch return FALSE is increased from 9 to 87. This allows the update of the global history register by the predicate define to complete before we fetch and predict the branches that are correlated with it. Using the region schedule in Figure 2(c) results in a global history register of {cmp1, cmp2, br3, br4}, and allows br3 to benefit from having cmp1 in its global history register during prediction. The optimal schedule would not move predicate defines as far away from branches as possible, but rather, just far enough to allow the predicate defines to update. For example, in the code shown in Figure 2(c) since we modeled a 20 instruction delay for predicate defines, the return FALSE instruction would be just as predictable if scheduled before the 40 instructions from block P5. This would reduce wasted dynamically executed instructions in the cases where return FALSE is taken.

6.2 Recovering the History State

An important topic in branch prediction is recovering the prediction history state after a branch misprediction. Since we are updating the global history register with branches in the fetch stage and predicate define instructions in the writeback stage, it is key that the updates to the speculative global history register occur in a consistent order for a given trace through the program's execution. In addition, we need to be able to correctly recover the speculative global history register in the case of a branch misprediction.

Our architecture uses a Speculative History Queue (SHQ) [15, 16] to hold the state of the global history register, so it can be restored on a misprediction. The SHQ stores the full speculative global history register into a queue each time a branch is predicted. When a branch is mispredicted the global history register for that branch is restored from the SHQ. The last bit in the history register representing the mispredicted branch is inverted, and this becomes the new speculative global history register used for prediction for the next fetch. For example, if branch br4 mispredicts in Figure 4, the speculative global history register will be restored to {br3, cmp1, cmp2, !br4}, correctly keeping track of cmp1 and cmp2.

In using the SHQ architecture as defined in [15, 16], there is a very small window in which a predicate define may not make its way into the SHQ, so that it can be restored after a misprediction. If the predicate definition resolves between the time a mispredicted branch is fetched and the time that branch triggers a misprediction, it will make its way into the speculative global history register, but not into the SHQ. This is because only a branch prediction will insert the speculative global history register into the SHQ. In this situation, the predicate define information will be lost if there is a misprediction before the next branch is fetched. We modeled this in our simulation results in the next section. The SHQ architecture could prevent the loss of this information by inserting into the restored global history register after the branch misprediction any predicate defines that would have been lost. Examining such an enhancement is left for future work.

7. EXPERIMENTAL EVALUATION

7.1 Methodology

We gather results for a subset of the SPEC95 benchmarks (go, gcc, m88ksim, and ijpeg), SPEC92 li, as well as two other benchmarks. Dot is a project from AT&T for plotting graphs, and gs is a run of ghostscript translating a paper from postscript to jpeg format. We chose these benchmarks because they have a reasonable number of mispredictions. We used the same input to the applications to generate the profile to guide region formation and to gather the misprediction results via simulation.

To gather our results we use a predicated form of the Alpha Instruction Set Architecture (ISA). We used the Alpha ISA, adding a predicate guarding register to every instruction, and we added predicate compare instructions to the ISA to define the predicates as shown in the examples earlier in this paper. We used this modified ISA to build predicated regions and to simulate the predicate scheduled code.

To conduct our experiments we used both ATOM [17] and SimpleScalar 3.0a [6]. We used SimpleScalar to calculate the average fetch to writeback latency for each branch instruction. This latency represents the number of cycles it takes to complete execution of a branch instruction from the time the branch instruction is fetched. We calculate this latency in SimpleScalar for each branch instruction and use it to build our own scaled down pipeline level simulator in ATOM in order to simulate the full execution of a program. For the SimpleScalar runs we simulate an 8-wide issue machine with a 128-entry RUU. The L1 data cache is 64K 4-way associative, L1 instruction cache is 32K 2-way associative, and we use a unified 1 MB 4-way L2 cache. The L1 miss and L2 hit latency is 12 cycles, with 120 cycle latency for an L1 and L2 miss. The minimum branch misprediction penalty is eight cycles, and we use a 32-entry return address stack for predicting return instructions.

To generate the predicate regions we used ATOM to profile the entire execution of the program to find the hard-to-predict branches. We then used the program analysis features of ATOM to provide an intermediate representation of the binary to schedule. We use this IR to form our predicated regions, as described in section 3, resulting in a new predicated representation of the binary. We also used ATOM to build a pipeline level branch simulator using the branch latencies from SimpleScalar, and simulate the predicated representation of the binary. This allows us to simulate the complete execution of a program, modeling the important branch latencies. We use the branch and predicate define latencies to model the out-of-order nature of predicate define updates to the speculative global history register, which occur once the predicate instruction resolves. We also use this to model the effect of recovering the branch

prediction state on a branch misprediction at the cycle the branch instruction finishes execution.

We evaluate our architecture by examining the improvements in branch misprediction rates, which are all normalized to the number of branch mispredictions in the original non-predicated code. That is, the misprediction "rate" is calculated as the number of mispredicts divided by the number of branch predictor accesses in the *original* execution of the program code. This allows us to look at one consistent metric as the number of branch predictor accesses will change with the predicated code. In addition, the miss rates we show include the misprediction rate for all branch types. This includes conditional branches, returns, unconditional, procedure calls, and indirect branches. In addition, we show the percent increase in instructions executed for the hard-to-predict predicate regions formed.

7.2 Baseline Prediction Results

We start by examining the percentage of mispredicted branches without using any predicate information to update the branch predictor. The first three bars in Figure 5 show the original mispredict rates using only local prediction, only global prediction, and the Meta Chooser (combination of local and global). We simulated a 4K entry local history, local pattern, global pattern, and chooser tables using a 12 bit local history and global history registers.

We show that the original Meta Chooser predictor performs better than either a predictor using just local (per-branch) information or one using just global information. The fourth bar in the graph shows the percent of mispredicted branches that occur after applying our region formation algorithm and predicating the hard-to-predict branches.

The results show that substantial reductions in miss rates are achieved for `go` (22% down to 15%), `ijpeg` (8.5% down to 2.5%) and `m88ksim` (3% down to 1%) using the hard-to-predict region formation method with a traditional branch predictor.

For the other programs, the results appear to show that we are not successful in attacking the misprediction problem with our hard-to-predict region formation approach. In reality (as we'll see in the detailed breakdown of mispredictions to follow) we are removing a large percent of mispredictions by translation of hard-to-predict branches to predicate defines, but the remaining region branches become much harder to predict using the baseline predicated Meta Chooser. Misprediction migration leads to disappointingly high misprediction rates, so much so that in `dot` and `li` very little decrease in overall misprediction rate is seen.

7.3 Predicate Global Update Predictor Results

The final four bars in the Figure 5 examine various methods of using predicate information in a Meta Chooser predictor and compare them to an idealized execution of the predicated code, where only branches executed on the true paths in regions have to be predicted. The details of each implementation follow. All predictors use a 4K-entry local history table to store local histories, a 4K-entry 2-bit pattern history table for local predictions, a 4K-entry 2-bit pattern history table and a 12-bit global history register for global predictions.

- Meta Chooser PEP: One of two local histories stored for the branch is used, based on the value of the guarding predicate register. The chosen history is the only one updated with the result of the branch. This provides results for the PEP predictor presented in [5].

- Meta Chooser Resolved PEP: We modified PEP to choose between its two local histories based on (1) knowing if the most recent predicate register definition for the guarding predicate is resolved or not, and (2) the value of the guarding predicate. If the most recent definition of the guarding predicate register has not resolved, then the false predicate local history is used to predict the branch. If it has resolved, than either the false or true local history is chosen based upon the value of the guarding predicate. The chosen history is the only one updated with the result of the branch. The results show that concentrating the true path local history on only those branches that have resolved provides a decent reduction in miss rate for a `li`.

- Meta Chooser PGU: The predicate define update of the speculative global history register is delayed until the predicate instruction has resolved. At this time its value is entered into the speculative global history register. Branches that are fetched after this that correlate with this predicate definition will benefit from having it in the global history register.

- True Path Only: For these results we model an environment where only those branches whose guarding predicate is true (i.e. whose path is live) are predicted and update history information. This isolates the effects of predicting branches guarded on a false predicate and provides an indication of what mispredict rate would be expected given the removal of the hard-to-predict branches via predication.

A Meta Chooser with Resolved PEP local predicate update averages a mispredict rate of 4.5% and one with PGU global predicate update averages a mispredict rate of 5%. Only having to predict branches that are guarded by a TRUE predicate, even with no predicate update, results in a miss rate of 2.75% on average.

7.4 The Impact of Falsely Guarded Branches

In Figure 6 we show a more detailed breakdown of the branch mispredicts in the original code, the code after predication, using Resolved PEP-style predicate information to affect local predictions, using the PGU predicate information to affect global predictions, and an idealized world where only true path branches have to be predicted.

The top portion of the original bars show the percent of mispredicts that will be removed from transforming those branches into predicate define statements when applying our hard-to-predict region formation algorithm. These results show that 44% to 91% of the original mispredicts in the programs are removed by if-converting these hard-to-predict branches.

The middle section of the original bars (True Path in Region) shows the mispredicts that come from region branches whose guarding predicate evaluates to true. The False Path in Region shows the percent of mispredicts caused by region branches that are guarded on false. The remaining mispredicts (in black) will lie outside of our predicated regions and will not be *directly* impacted by our predictor modifications,

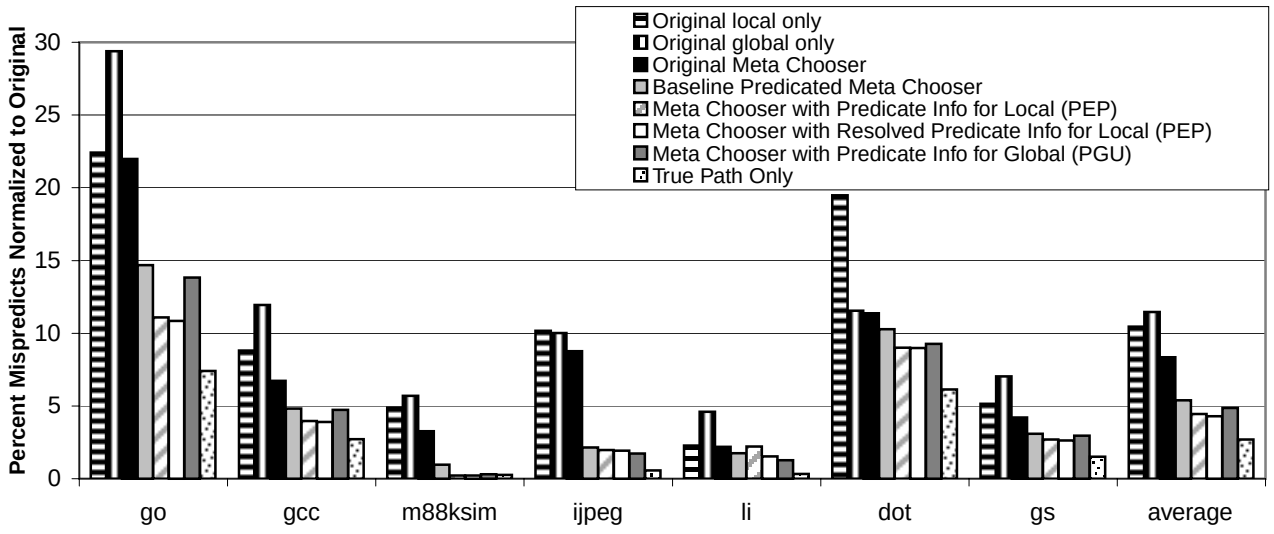


Figure 5: Change in misprediction rate, normalized to the number of predictor accesses in the original unpredicated code.

but may be affected by the global history update from predicated regions.

The number of dynamically fetched branches for predicated code where only the true path branches are fetched is reduced between 23% (for `li`) and 50% (for `ijpeg`). However, when considering the spurious (false path) branches that must be fetched in predicated regions, the number of dynamically fetched branches is not always reduced compared to the original code. Three benchmarks still have overall reductions in dynamically fetched branches (`m88ksim` , `ijpeg`, and `dot`), two have almost the same number of predictions as in the original code (`go` and `gcc`), and the rest see an *increase* in the number of dynamically fetched branches.

In Figure 6, across all the predicated results, we see little or no decrease in mispredicts for non-region branches and a marked increase in the number of mispredicts caused by region branches. Additionally, while false path mispredictions are significant in and of themselves, by comparing true path only results with the other predicated results, we see that false path (spurious) predictions and mispredictions have a marked impact on the mispredicts caused by true path region branches.

7.5 Squash-FP to Remove False Branches

In Figure 7 we show various branch prediction schemes used to minimize the negative impact of falsely guarded branches on region branch prediction. Squash-FP uses information from the predicate register file to determine if a branch's guarding predicate is false and then effectively squash the branch by predicting it as "not-taken". The Squash-FP prediction filter is only used if the most recent definition for the guarding predicate has resolved. Squash-FP requires knowing the guarding predicate register for each branch, which we assume is saved per branch in the BTB.

Figure 7 shows six results – 4 using Squash-FP to first filter out the branches that are guarded on false predicates, where the predicates have resolved by the time the branch is predicted. The implementations shown are:

- Baseline Meta Chooser: Uses no predicate information, must predict all region branches.
- Squash-FP Baseline Meta Chooser: Uses guarding predicate information as stored per branch in the BTB. Predict not-taken for branches whose guarding predicate is resolved and has a value of false.
- Squash-FP PEP Meta Chooser: If Squash-FP does not apply to the branch, then use the predicate value from the predicate register file to select between the two per branch local histories. Use this local history if local history is chosen by the meta chooser, otherwise use the default global predictor.
- Squash-FP PGU Meta Chooser: If Squash-FP does not apply to the branch, then use PGU to predict the branch if global prediction is chosen by the meta chooser. All branches update the global history register, otherwise use the default local predictor.
- PEP+PGU Meta Chooser: Uses PEP-style predicate information for local predictions and PGU-style for global predictions.
- Squash-FP PEP+PGU Meta Chooser: If Squash-FP does not apply to the branch, uses PEP-style predicate information for local predictions and PGU-style for global predictions.

Each bar in the graph is broken into five sections indicating the source of the misprediction. The bottom section shows misses from Non Region areas. The next two show misses from true path branches in regions - the stripes are from prediction where the value of the guarding predicate was known to be true at the time of prediction. The top two show misses from false path branches - the stripes are from predictions where the value of the predicate was resolved false (these predictions are squashed via Squash-FP).

The four implementations of Squash-FP show a clear benefit over the other two in that all misses from resolved false

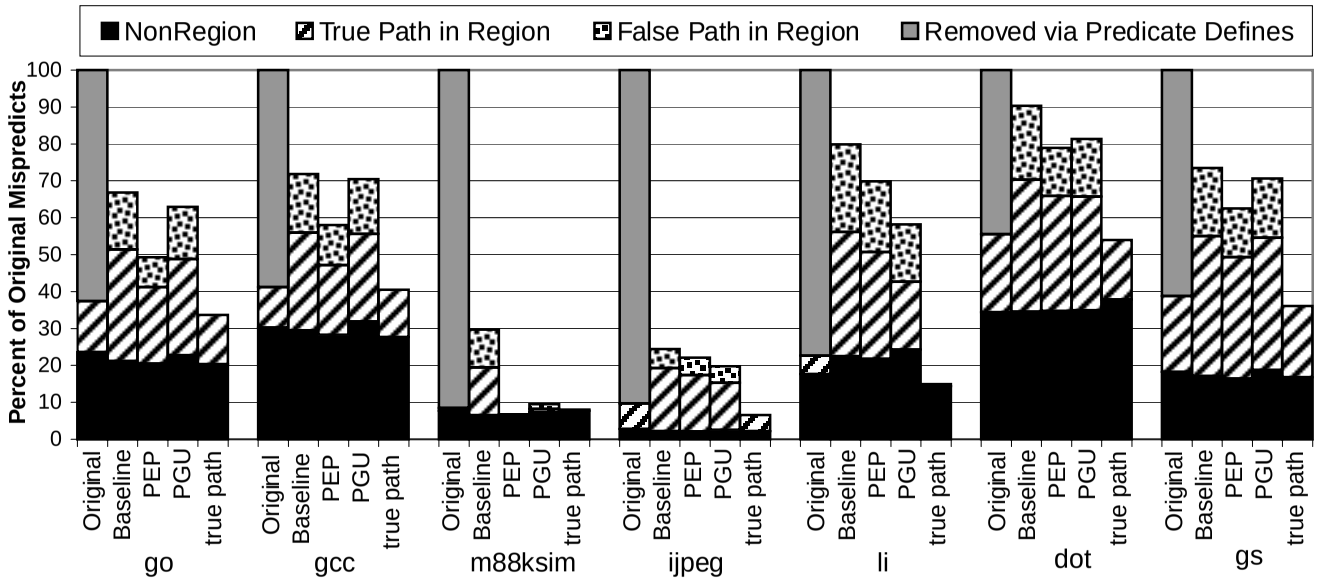


Figure 6: Comparison of the breakdown of locations of mispredicts in the program, normalized to the original number of mispredicts per benchmark. In each set of bars we show information (from left to right) for the original non-predicated code, the baseline Meta Chooser predictor on the predicated code, Meta Chooser using PEP update for local predictions, Meta Chooser using PGU for global predictions, and an idealized execution of only the true path of the predicated code.

path predictions (dark stripes) are removed. Some benchmarks like `gcc` and `gs` do better when incorporating predicate information into the local predictor. Some benchmarks do better when incorporating predicate information into the global history register. The latter can be particularly beneficial when the full guarding predicate of a branch is not resolved at prediction. `li` shows significant benefit from PGU update which we attribute to its very low percent of resolved predicates (as discussed in Section 7.6). Some benchmarks like `go` , `gcc`, and `m88ksim` achieve as much benefit from using predicate information to just reduce resolved false path predictions as from further predicate usage in either PEP or PGU. On average, PEP sees very little improvement with Squash-FP, PGU sees .6% improvement with Squash-FP, and PEP+PGU sees .1% improvement. A Baseline predictor that only uses predicate define information to do Squash-FP improves its misprediction rate by a full 1% with this spurious branch detection optimization.

7.6 Increase in Executed Instructions

We applied our region formation algorithm in section 3 to form predicated regions for the top 10% most frequently mispredicting branches in each benchmark. For the SPEC95 program `go`, 87% of all mispredicts in the program can be attributed to the top 10% most frequently mispredicting static branches. Additionally, we aggressively scheduled predicated regions trying to separate predicate defines and branches at the cost of execution of extra code. As stated earlier, we did not try to limit the increase in executed code since our study focused on examining the maximum reduction in branch misprediction rate achievable using the techniques presented in this paper. Dynamic instruction counts of the predicated versions of the codes increased across a wide range from 10% to 94% as shown in Figure 8.

The regions we formed provided a large range of exploration with regards to predicate define and branch scheduling placement. Scheduling of branches and the predicate defines that guard them is paramount as most of the predicate update predictor techniques evaluated rely on information from "resolved" branches – ie, those branches whose guarding predicates have been defined by the time the branch is fetched. Our region formation lead to widely ranging numbers of branches whose guarding predicate is resolved: in `li` only 7.5% of all region branches have their guarding predicate resolved at fetch, while in `m88ksim` all region branches have their guarding predicate resolved.

In future work, we will be examining the effects that this region formation has on performance and refining our region formation algorithm to tradeoff the increase in code with the reduction in branch miss rate.

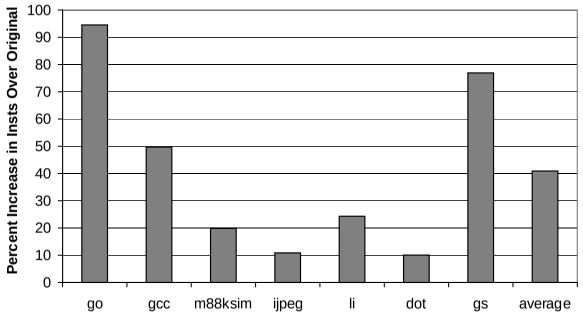


Figure 8: Increase in dynamic instruction count from our hard-to-predict predicate region formation and scheduling techniques.

7.7 Variability of Predicate Define Updates

Due to the delayed update of predicate define information, any scheme which utilizes predicate define information

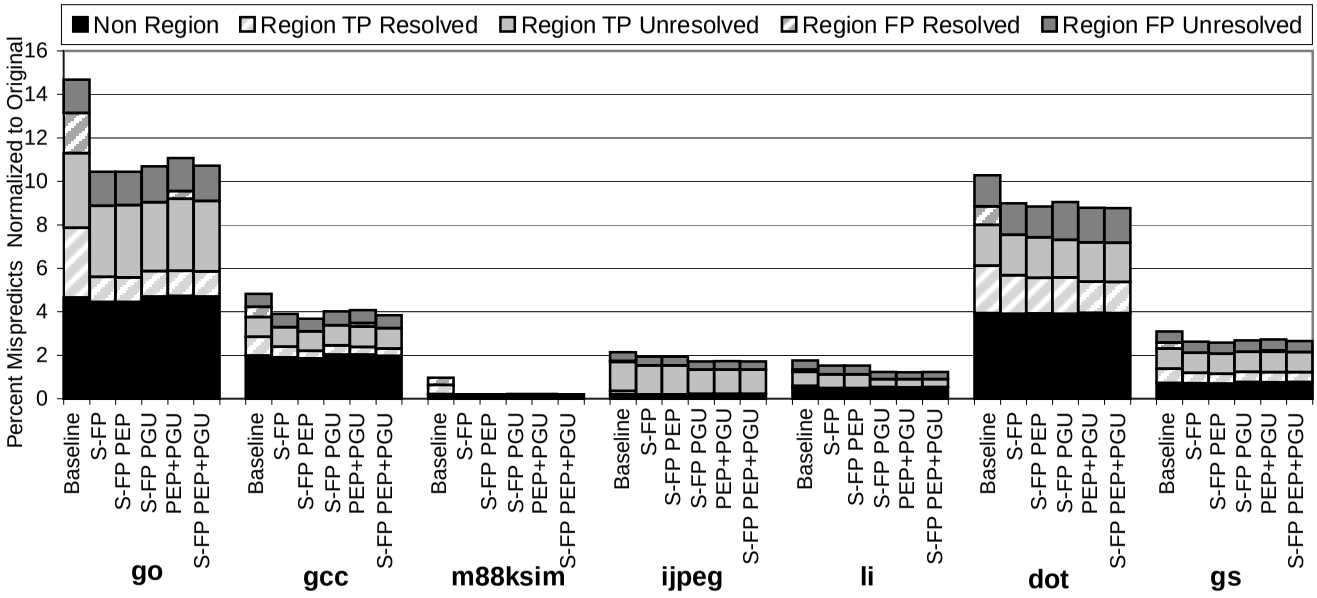


Figure 7: Change in misprediction rate, normalized to the number of predictor accesses in the original unpredicated code. Columns 1 and 5 of each group are not optimized with Squash-FP, the rest show the benefit gained by removing resolved false path predictions.

in conjunction with a global history register is problematic given variability in the latency of predicate define instructions.

With the PGU scheme, if a predicate define uses an operand whose load occasionally misses in the cache, then some dynamic occurrences of branches fetched shortly after the predicate define will see "different" global history registers. In cases where the load does not miss, the predicate define may reach commit and update the global history. In cases where the load misses, that predicate define may not update the global history, yet other fetched branches may update the global history. This variability in ordering of global history update will have a negative effect. Our results currently use a fixed length predicate define latency determined for each branch from SimpleScalar. The only differences in global history update sequences we see occur when a branch mispredicts and additional cycles are available (from the mispredicted path) for predicate defines to update the global history register. For Squash-FP, we always update the global history of a region branch, even if its predicate is resolved false and its prediction is squashed. This way, while our ability to determine that a branch is guarded on false might differ based on system effects, the global history register entries will not suffer from variance.

While this effect might be relatively small in an in-order processor like the Itanium, the issue will certainly be more noticeable in an out-of-order implementation. One possibility would be to fix a delay time for each static predicate define. Then, as the predicate define was fetched, an entry would be made in a structure that would be decremented every cycle – effectively acting as a timer. When a predicate define's timer went off, if the define had resolved, the resolved value would update the GHR. If not, the GHR would update with false. This would guarantee fixed update of the GHR and would allow the compiler flexibility in the update of the GHR with predicate define information. We are examining the performance of this design as part of future work.

7.8 Comparing PEP and PGU

The PEP predictor uses the predicate information to choose which local history prediction to use, whereas our PGU predictor includes the predicate information into the global history register.

For the results presented, we examined both architectures using roughly the same area. Comparing the complexity of these two designs in terms of access time, the PEP architecture will be difficult to implement in a single cycle. The critical path of the PEP architecture requires a two table lookup to perform a conditional prediction, whereas the PGU architecture requires only a one table lookup to perform its conditional branch prediction. The PEP architecture first needs to look into the BTB to find out which predicate corresponds to the branch and the two local histories. It then looks up, in parallel, the predicate register file to obtain the latest value for that predicate and the 2-bit predictors for the two local histories. Only then can it choose its prediction using the result of the register file lookup. In comparison our PGU architecture performs only one table lookup by using the global history register to index into a 2-bit table of counters, similar to existing branch prediction architectures. The reduced complexity and access time of the PGU design makes it an attractive option for implementing a predicate-aware branch predictors.

8. SUMMARY

Predication allows hard-to-predict branches to be removed and replaced with predicate defines, which do not have to be predicted. In order to effectively reduce branch predictions, predicated region formation must focus on the most offending hard-to-predict branches. To do this we found that we had to allow unbiased, though originally predictable,

branches to reside in predicated regions. Therefore, we developed a hard-to-predict region formation algorithm targeted at removing many of these hard-to-predict branches, while allowing some unbiased, predictable branches to remain in the predicated region. On average, predicate region formation reduced the branch mispredict rate from from 8% to 5.5% across the benchmarks when using our hard-to-predict region formation.

Without any modification to branch prediction hardware, region branches become a major problem in achieving the reduced branch misprediction rates we expect from predicated codes. The ability to accurately predict these region branches is hindered by their increased dynamic occurrence, their new prediction pattern based on their guarding predicate dependences, and the fact that information from predicate define statements is no longer available in the global history register.

To address this problem, previous work has investigated augmenting local per-branch prediction schemes with guarding predicate information. We propose a complementary Predicate Global Update Branch Predictor architecture to improve the prediction of region branches with the goal of reducing misprediction migration. Our Predicate Global Update Branch Predictor allows predicate define statements to provide correlative information to the branch predictor state by updating the speculative global history register at writeback. We model updating the speculative global history register out of order using predicate define statements, and recovering the state of the branch predictor on a branch misprediction.

As much as possible, we schedule regions to allow predicate define information to be utilized by both of the predicate sensitive prediction schemes.

Finally, we propose a direct approach to the problem of branches whose guarding predicate is false using Squash-FP. Squash-FP achieves 100% prediction accuracy for spurious branches whose guarding predicate definitions have resolved by the time the branch is fetched. These falsely guarded branches should always predict "not-taken". This technique uses a branch-to-guarding predicate identification scheme via the BTB similar to PEP. However, it takes advantage of bypass information stored in the architecture pipeline to determine when branches are guaranteed to be spurious. This technique also has the benefit of reducing contention in the branch prediction tables by no longer using predictor state that was previously used by false path branches.

We show that, even alone, the Squash-FP method of utilizing predicate define information achieves a sizable reduction in branch mispredictions (ranging from .5% to 4.3%). This method is arguably the simplest and fastest predicate update modification to current branch prediction architectures. Squash-FP can also be employed with PEP, PGU, or a Meta Chooser predictor utilizing both PEP and PGU. The benefit measured with these techniques is modest on the average, but individual benchmarks experience significant improvements.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded by NSF grant No. CCR-0073551, and a grant and equipment donation from Intel Corporation.

9. REFERENCES

[1] Intel Corporation: Itanium Processor Architecture. http://www.intel.com/design/ia-64/index.htm.

[2] *Intel IA-64 Architecture Software Developer's Manual, Volume 3: Instruction Set Reference.* January 2000.

[3] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.

[4] D. I. August, W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[5] David I. August, Daniel A. Connors, John C. Gyllenhaal, and Wen mei W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *The 3rd International Symposium on High-Performance Computer Architecture*, pages 84–93, 1997.

[6] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[7] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *Proceedings of the 34th Annual Intl. Symp. on Microarchitecture*, December 2001.

[8] L. Gwennap. Intel, HP make EPIC disclosure. *Microprocessor Report*, 11(14):1–9, October 1997.

[9] R.E. Kessler, E.J. McLellan, and D.A. Webb. The alpha 21264 microprosessor architecture. In *International Conference on Computer Design*, December 1998.

[10] Artur Klauser, Todd Austin, Dirk Grunwald, and Brad Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 18th Annual International Conference on Parallel Architectures and Compilation Techniques*, pages 278–285, 1998.

[11] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th Annual Intl. Symp. on Microarchitecture*, pages 217–227, December 1994.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual Intl. Symp. on Microarchitecture*, pages 45–54, December 1992.

[13] Scott A. Mahlke and Balas K. Natarajan. Compiler synthesized dynamic branch prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 153–164, 1996.

[14] J. C. H. Park and M. Schlansker. On Predicated Execution. Technical Report HPL-91-58, HP Labs, May 1991.

[15] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. In *IEEE Transactions on Computers*, April 2001.

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

[17] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. Association for Computing Machinery, 1994.

[18] G. S. Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 196–206, November 30–December 2, 1994.