Incorporating Predicate Information Into Branch Predictors

Beth Simon[†] Brad Calder[‡] Jeanne Ferrante[‡]

[†]Department of Mathematics and Computer Science, University of San Diego
[‡] Department of Computer Science and Engineering, University of California, San Diego
{bsimon}@sandiego.edu, {calder,ferrante}@cs.ucsd.edu

Abstract

Predicated Execution can be used to alleviate the costs associated with frequently mispredicted branches. This is accomplished by trading the cost of a mispredicted branch for execution of both paths following the conditional branch.

In this paper we examine two enhancements for branch prediction in the presence of predicated code. Both of the techniques use recently calculated predicate definitions to provide a more intelligent branch prediction. The first branch predictor, called the Squash False Path Filter, recognizes fetched branches known to be guarded with a false predicate and predicts them as not-taken with 100% accuracy. The second technique, called the Predicate Global Update branch predictor, improves prediction by incorporating recent predicate information into the branch predictor. We use these techniques to aid the prediction of region-based branches. A region-based branch is a branch that is left in a predicated region of code. A region-based branch may be correlated with predicate definitions in the region in addition to those that define the branch's guarding predicate.

1. Introduction

The Explicitly Parallel Instruction Computing (EPIC) architecture has been put forth as an architecture for achieving the *instruction level parallelism* (ILP) needed to keep increasing future processor performance [6]. The IA-64 Itanium processor [7] 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. [5] 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 may have a much larger impact.

The goal of this paper is to examine advances in branch prediction for predicated instruction sets, not a new nor an optimal region formation algorithm. To this end we focused on a region formation that is aggressive at removing hard-to-predict branches, and the branch predictors we examine concentrate on predicting the remaining branches. In order to aggressively form predicated regions around 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. A region branch will now be predicated on a predicate 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 the guarding predicate is TRUE, and should always be predicted as not-taken when the guarding predicate is FALSE.

In this paper, we develop two new branch predictor optimizations and evaluate them for an in-order EPIC processor. First is a new branch prediction optimization called *Squash False Path* (Squash-FP) that attempts to know, at fetch, a branch's guarding predicate value and, if it is false, the branch is predicted as not-taken. The goal of this predictor is to correctly predict region branches that are on the false path as not-taken.

The second predictor we developed adds predicate informa-

tion into the global history register. We examine the *Predicate Global Update Branch Predictor* (PGU) architecture to incorporate predicate information into the global history to try to improve the performance of region branches that benefit from correlation. The PGU updates the global history non-speculatively with the predicate result when the predicate defining instruction is resolved. In a PGU predictor, correct predicate information from true-path predicates is placed in the GHR. This can allow region branches to benefit from this history correlation when making their prediction from the global prediction table. We propose the *Deterministic Update Table* (DUT) that provides a reproducible GHR update for predicate definitions to allow the PGU to achieve a lower miss rate.

For the non-speculative predicate-aware architectures (Squash-FP and PGU), region branches only benefit 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.

We compare the performance of Squash-FP and PGU to the Predicate Enhanced Prediction of August et. al. [1]. Their predictor incorporates predicate information into local per-branch predictors. We examine the performance of their predictor and hybrid combinations of all three techniques. In addition, we compare the performance of PGU to a branch predictor that predicts predicates during fetch and inserts the speculative predicate predictions into the global history register. We show that this approach performs worse than just using the default branch prediction architecture for predicated code.

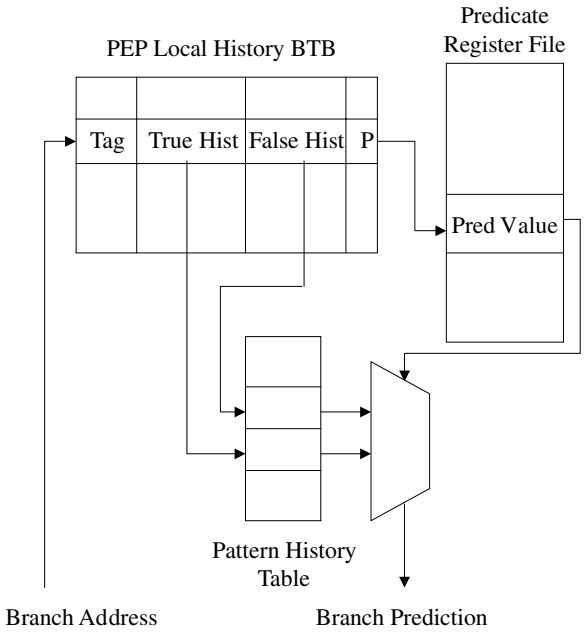


Figure 1: High level design of the Predicate Enhanced Prediction Architecture.

2. Prior Work

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, 3, 2]. 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.

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 [17] 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 [17], that branches left in these idealized regions are only predicted when their guarding predicate is true.

In [1], 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 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.

In other related work, 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 store the result of this function in a predicate register, which is then used in a modified version of the prepare-to-branch instruction that precedes a branch in their architecture.

3. Predicated Region Formation

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

The goal of this paper is to examine advances in branch prediction architecture for predicated instruction sets, not a new nor optimal region formation algorithm. To this end we focused on a region formation algorithm that aggressively removes hard-to-predict branches and the branch predictors we examine concentrate on predicting the remaining branches.

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 7.8.

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.

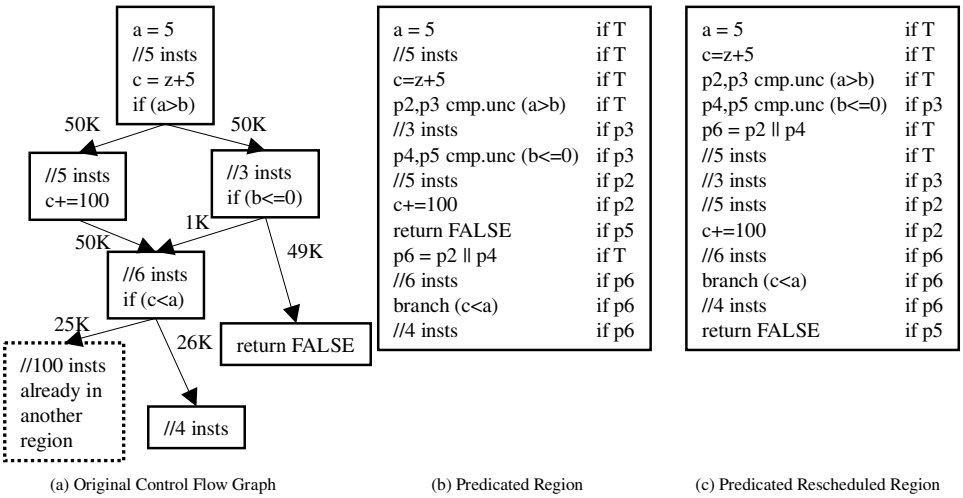
There are additional measures we use in controlling region formation. First any successor block that is reached less than 10% of the time the region is entered is excluded from the region. This keeps cold blocks from unnecessarily bloating the region with infrequently useful work. Second, any block ending 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)` in Figure 2(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 actu-



(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)`.

ally 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 entering 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 followed. In Figure 2(b) branch `return FALSE` will be fetched and have to be predicted 100K times, approximately twice as frequently as in the non-predicated code. This impacts the predictability of these instructions and can significantly increase the number of 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 (`(c<a)` 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. Figure 2(c) shows a rescheduling of the code such that `p5` has more time to complete execution before the return branch is predicted. 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 `(a>b)` 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 4K entry local, global, and chooser tables using a 12 bit global history register. This 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 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 global history register and the local history register in fetch. They 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. [1]. 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 propose 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 2-bit 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. We use the Squash-FP as a filter for any type of branch predictor, filtering out branches with this property at fetch, predicting them as not-taken. A branch predicted as not-taken this way does not update the 2-bit history table, but it does insert its information into the global history register.

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 Predictors

We will now examine incorporating predicate information into the global history of a branch predictor. In both predicate-aware architectures presented below, region branches (both spurious and true-path) update the predictors in exactly the same fashion as the baseline predicated branch predictor. All region branches are predicted and speculatively update the global history register in fetch. In [15], they showed that the global history should be speculatively updated to achieve decent performance, and the history is recovered on a misprediction.

6.1 Speculative Predicate Update

The most straight-forward way of incorporating predicate information into a branch prediction scheme is to follow the same update model as branches – predict the result of predicate definitions and speculatively update the GHR with predicate defines in fetch. In this scheme, called Speculative Predicate Update (SPU), predicate defines must be "predicted" during fetch and the result of that prediction speculatively updates the branch predictor structures. However, there is no penalty for "mispredicted" predicate defines. The predicted information is only used to feed the branch predictor, not to control fetch or other pipeline functionality as occurs with branches.

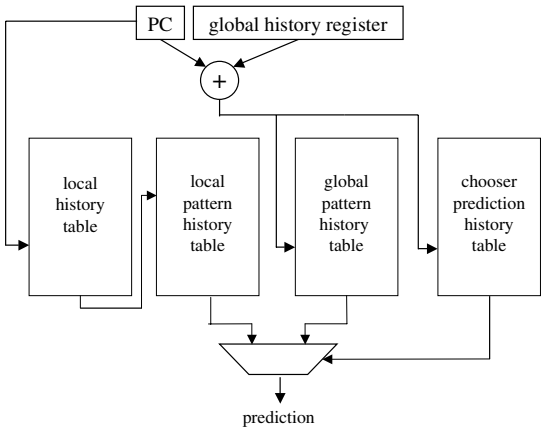


Figure 3: Our baseline Meta Chooser branch prediction architecture. It contains a local history predictor, global history predictor, and a 2-bit chooser table to choose between the two predictions.

A negative effect of the prediction of predicate defines at fetch comes from the need to predict both true-path and false-path predicate defines. Although predicate defines guarded on false predicates never set a predicate value to true, the SPU must make predictions in fetch – before the value of the guarding predicate may be known. This will increase the capacity concerns of branch prediction structures (as all predicate defines are predicted and update the branch predictor) and may contribute to dilution of useful information in the GHR in cases where many false-path predicate defines are fetched.

Our results show that speculatively inserting predicate definitions into the GHR has a higher miss rate than not including them at all. A similar result was found in [10], where they dynamically predicated small hammock branches.

6.2 Resolved Predicate Global Update

The goal of our *Predicate Global Update* (PGU) predictor architecture is to capture the *result* of true-path predicate define statements in the global history. Ideally, we want to do this after the predicate instruction finishes execution in the writeback stage as this allows us to avoid the process of predicting predicate values and enables us to update with only true-path predicate defines. However, this will cause a "delay" in the update of the global history register – which may reduce the usefulness of the predicate information. This can be addressed through code scheduling techniques and is examined further below.

6.2.1 Basic PGU Example

Combining delayed (non-speculative) GHR update by predicate defines with speculative GHR update by branches will result in a different ordering of global history information than in the original non-predicated program. We show how this "reordering" and the delayed, non-speculative update by pred-

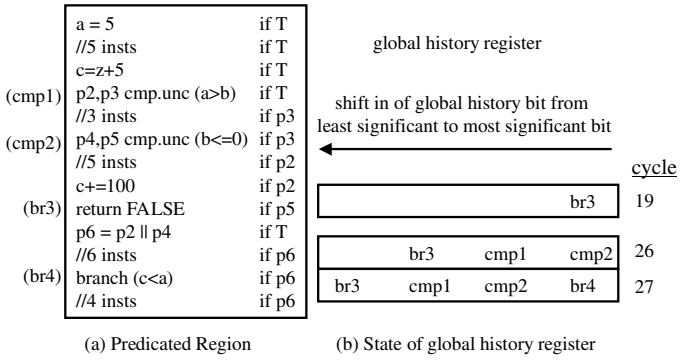


Figure 4: Update of the Meta Chooser global history register when enabled with predicate update. Note that branches update the global history register in fetch, while predicate defining instructions update it in writeback. This causes a "re-ordering" of information in the global history register as compared with instruction fetch ordering.

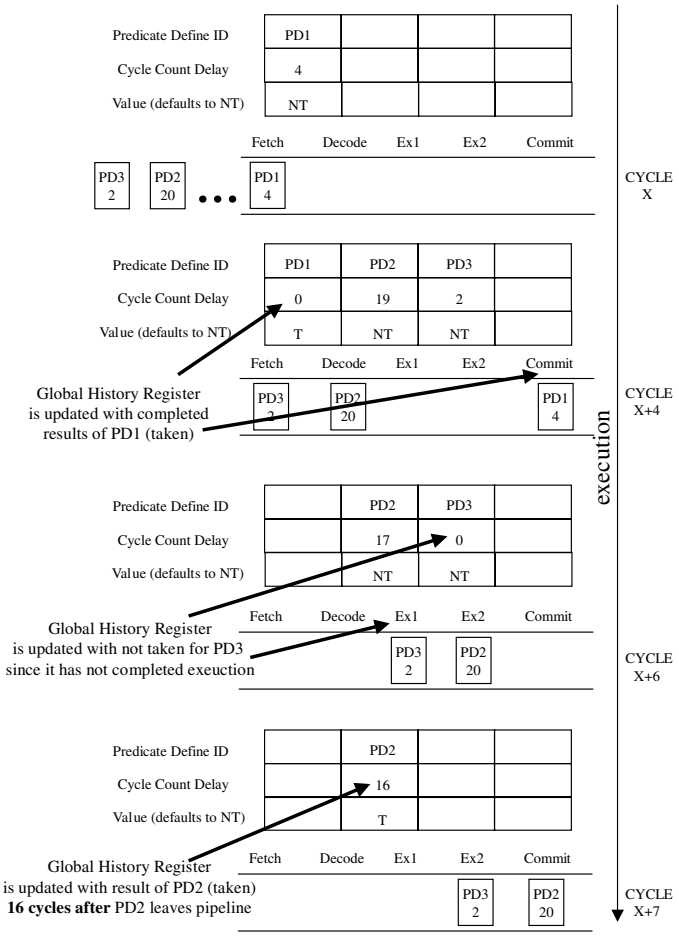


Figure 5: A Deterministic Update Table (DUT) to ensure a deterministic ordering of branch and predicate define (PD) information seen by the GHR. Each entry contains a PD ID, delay, and result value. The delay acts as a timer, and upon reaching zero, the Value entry is inserted into the GHR. Result values are initialized to not-taken `NT` and updated when the PD finishes execution. If a PD's delay expires before it finishes execution, the GHR is updated with the default `NT` value.

icate defines can impact our ability to benefit from predicate define information.

Figure 4 shows an example of how non-speculative predicate define update impacts the history stored in the global history register. Note that the fetch order of instructions doesn't match the GHR update order. The GHR is updated by a bit shift from the right (into the least significant bit). For clarity in this example, we assume each predicate define doesn't finish execution until 12 subsequent instructions are fetched.

Using the predicate update branch predictor, the global history register is updated with the values of predicate registers `p2` (cmp1) and `p4` (cmp2) when they are resolved. The predicate defines formed via our region formation process all define two complementary predicates, and we model the architecture such that it updates the 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 [8]. 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 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.

However, 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 `cmp`s is updated in time. One 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 `cmp2` and branch `return FALSE` is increased from 6 to 21. 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, br4, br3`}, and allows `br3` to benefit from having `cmp2` 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 the GHR before the branch is fetched. For example, in the code shown in Figure 2(c) since we illustrated a 12 instruction delay for predicate defines, the `return FALSE` instruction would be just as predictable if scheduled before the instructions from block `p6`. This would reduce wasted dynamically executed instructions in the cases where `return FALSE` is taken.

6.3 A Deterministic Predicate Update Architecture

A consistent dynamic ordering of information in the global history register is paramount for achieving global branch pre-

diction accuracy. PGU's multiple-pipeline-point update procedure for the GHR exposes the *update reproducibility problem* – even for in-order processors. Between the time a predicate define is fetched and has completed execution, the define may stall in the pipeline due to system-level effects such as dependency on a cache miss. If the predicate define is detained from completing execution by a stalling instruction, fetch can continue, simultaneously filling up the in-order instruction window. If, during the stall, a branch is fetched, it will be predicted and speculatively update the GHR, after which the predicate define may finish execution and complete its update of the GHR. On another execution of this same instruction stream, the predicate define may not be stalled and will complete execution (and update) *before* that branch was fetched. This situation could arise in the code shown in Figure 4. In the case where one of the 3 instructions guarded by `p3` sometimes causes a pipeline stall, we may get a GHR of the ordering {`br3, cmp1, cmp2, br4`} when there is no stall and one of the ordering {`br3, cmp1, br4, cmp2`} when an instruction guarded by `p3` stalls the pipeline.

This means that the ordering of information in the global history register may not be dynamically reproducible with respect to a given branch, because our architecture can update the GHR from multiple points in the pipeline (fetch and write-back). To address this problem we present an architecture that provides a deterministic update to ensure a reproducible global history register for a given path, while at the same time trying to update the GHR with results from completed instructions.

We propose a *deterministic predicate update table* (DUT) in the processor architecture to guarantee a dynamically reproducible history in the face of delayed predicate define update of the global history. Figure 5 shows a schematic of our structure. With this structure, we fix a delay time for each static predicate define to update the global history – whether or not the dynamic occurrence of the predicate define has completed execution.

This structure is not limited to gathering information from predicate defines. Any instruction type could be tagged for update into the DUT with a delay representing when that information is likely to be available. In this work, we only examine the use of the DUT for incorporating predicate define results.

In the DUT we store the following information for each predicate define as it is fetched: predicate define identifier (PD ID), delay to update (in cycles), and result value. The predicate define ID is the ID used by the processor to denote the dynamic definition of the instruction defining that predicate. Result value is the value used to update the global history for this predicate define – e.g. the result of the predicate define instruction. Before the instruction has completed execution, this is set to a default value of "not-taken". The delay to update determines when the value stored in result will update the global history register. This delay is initialized to a particular value for each predicate define relative to the cycle in which the predicate define instruction is fetched. Every cycle thereafter, the counter is decremented. When the value reaches zero, the result value is stored into the global history register providing a deterministic ordering for the global history. If a predicate define finishes execution before the delay counter triggers, the result of the predicate define is stored in the result field. In this case the update of the global history will be accurate (i.e., result contains the actual result of the execution of the predicate define). Either way, this architecture guarantees a reproducible order of including branch/predicate define information in the global history.

For a branch to benefit from predicate information stored in the global history register, predicates with which it correlates need to update the history before that branch is fetched and predicted. The delay time used for this architecture can be implemented in many different ways. Some of the possibilities include having a fixed time for all predicate defines, calculating delay via profiling or compiler analysis and, for each instruction, specifying the delay in the ISA, or calculating delay dynamically on the fly. For our simulation results, we profiled the average delay between fetch and execution completion for each predicate define instruction using SimpleScalar. We then used the corresponding delay for each predicate instruction in our in-order simulator for the DUT.

In Figure 5 we show how the DUT performs when processing three different predicate define instructions. In (a) the first predicate define, `PD1`, is fetched and inserted into the DUT with a delay of four cycles. Next `PD2` is inserted with a delay of 20 cycles. Then, in cycle `X+4` `PD3` is inserted with a delay of 2 cycles *and* the cycle counter for `PD1` has been decremented to zero. At this point, the global history register is updated with the value stored in `PD1`'s value entry - which was updated after `PD1` finished execution with the result of `taken`. After two more cycles have elapsed, the cycle count for `PD3` reaches zero and it updates the global history register with the value in the DUT. As `PD3` has not finished execution in the pipeline, the update to the global history register will be the default of `not-taken`. There is no other update of the global history register made when `PD3` finishes execution. In this case, it is possible that we have entered invalid data for `PD3` since we updated the global history before PD3 completed execution. Finally, even though `PD2` commits in the pipeline in cycle `X+7`, it will not have its result value inserted into the global history register until cycle `x+23` - 16 cycles after it leaves the pipeline. The result of its execution is stored in the DUT's value entry for `PD2` until then and is used in updating the global history register.

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 conduct our experiments we used both ATOM [16] and SimpleScalar 3.0a [4].

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 predicated, scheduled code. 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 the predicated regions. We used SimpleScalar to calculate latencies to create an in-order simulator to simulate our Alpha predicated code. We did this, because it was easier than modifying SimpleScalar to consume our Alpha predicated binaries.

For the SimpleScalar runs to generate the latencies for our branch centric ATOM simulator, 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.

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.

For the Speculative Predictor Update (SPU) (where predicate define instructions are predicted) we do *not* count predicate define mispredictions in the mispredict rate as their misprediction has no cost (no architectural penalty). 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 6 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 *fifth* 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 Speculative Predicate Update Predictor Results

The fourth bar in the graph shows the effect of speculatively updating the branch predictor with predicted predicate define information. For all benchmarks, this results in more branch mispredictions than the Baseline Predicated Meta Chooser (fifth bar) which does not attempt to incorporate predicate define information into the branch predictor. Here, even though predicate information is updated in a "timely" manner in fetch, the prediction rate suffers due to two effects. First, all predicate defines (even spurious predicate defines guarded by a false predicate) speculatively update the predictor. This can dilute the GHR in cases where many false path predicate defines are fetched. Secondly, predicate defines are frequently hard to predict. Often, this is one of the factors that influenced the decision to translate a branch into a predicate define in region formation.

7.4 Predicate Update Predictor Results

The last four bars in the Figure 6 examine the various competitive 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 [1].

- Meta Chooser Resolved PEP: We improved the PEP architecture 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 global history register is delayed using the Deterministic Update Table as described in Section 6.3. When the delay is triggered its value is entered into the global history register. Branches fetched later that correlate with

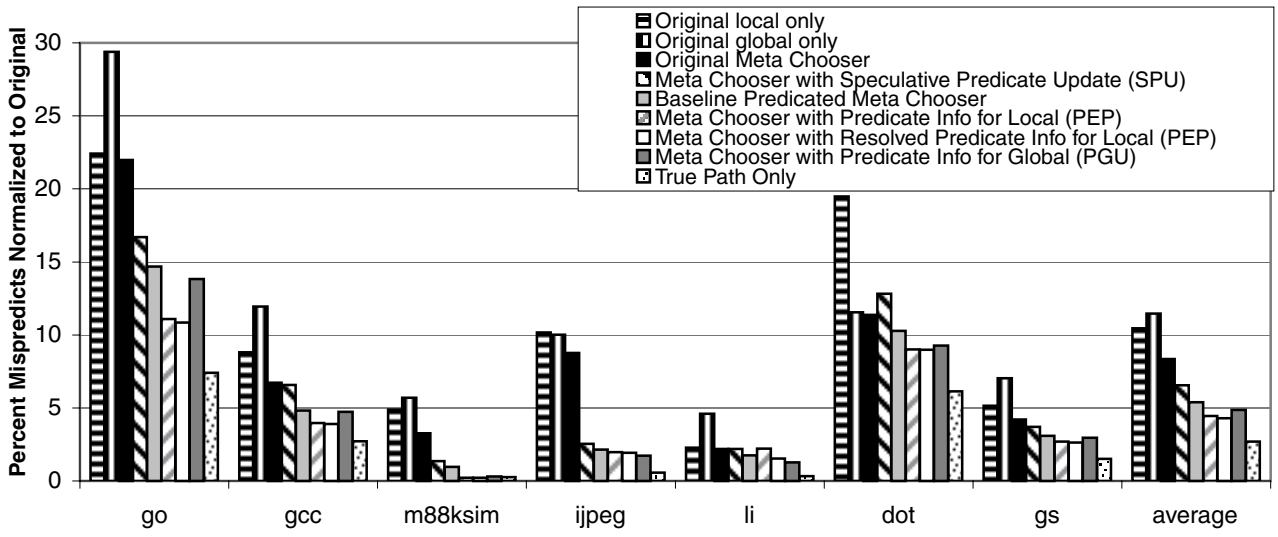


Figure 6: Change in misprediction rate, normalized to the number of predictor accesses in the original unpredicated code.

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.

7.5 The Impact of Falsely Guarded Branches

In Figure 7 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 should be removed by 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, 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 7, 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, we see that false path (spurious) predictions and mispredictions also have a marked impact on the mispredicts caused by *true path* region branches. This is shown in the increase of true path mispredictions (striped bars) in any "realistic" prediction scheme as compared to the predictions from true path branches in the "true path only" results.

7.6 Squash-FP to Remove False Branches

In Figure 8 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 know-

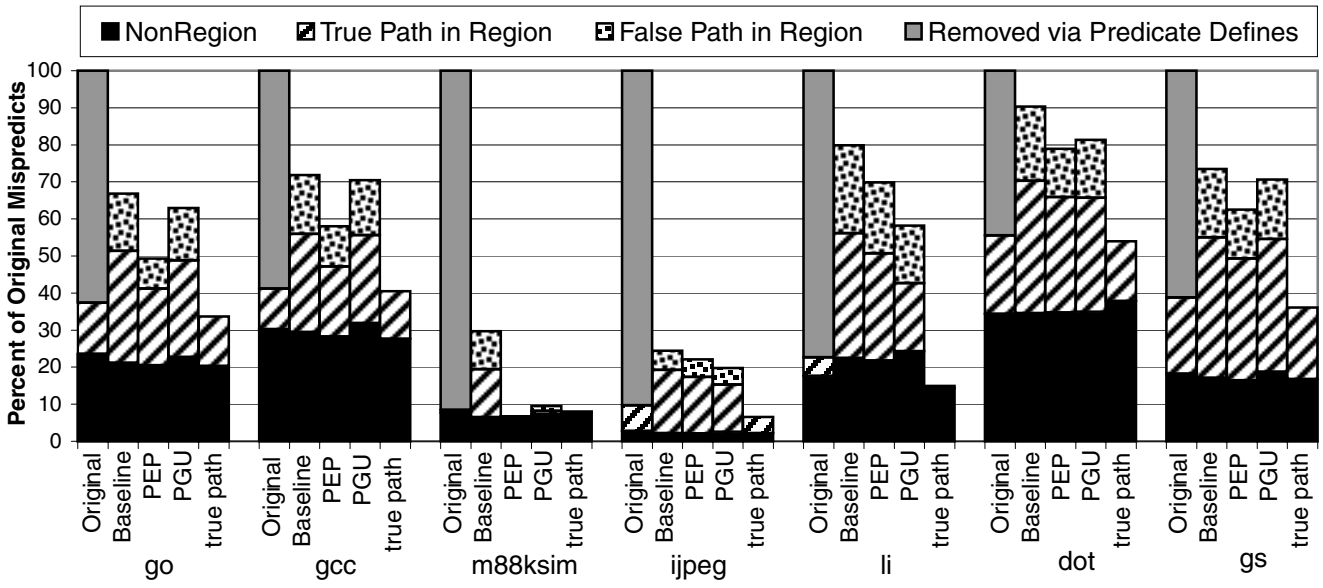


Figure 7: Comparison of the breakdown of locations of mispredicts in the program, normalized to the original number of mispredicts per benchmark. In each set of bars we show information (from left to right) for the original non-predicated code, the baseline Meta Chooser predictor on the predicated code, 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.

ing the guarding predicate register for each branch, which we assume is saved per branch in the BTB.

Figure 8 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, otherwise use the default baseline local predictor. All branches update the global history register.
- 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 path predictions (dark stripes) are removed. The results show that adding the Squash-FP filter to the baseline predictor achieves the majority of reduction in branch miss rate. 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 7.8). In comparing Figure 6 with Figure 8 PEP sees on average very little improvement with the Squash-FP filter, PGU sees 0.6% improvement with Squash-FP, and PEP+PGU sees 0.1% improvement. A Baseline predictor that only uses predicate define information to implement Squash-FP improves its misprediction rate by a full 1%.

7.7 Timing Discussion

For the results presented, we examine each architecture using roughly the same area. Comparing the complexity of the designs in terms of access time, both the PEP and Squash-

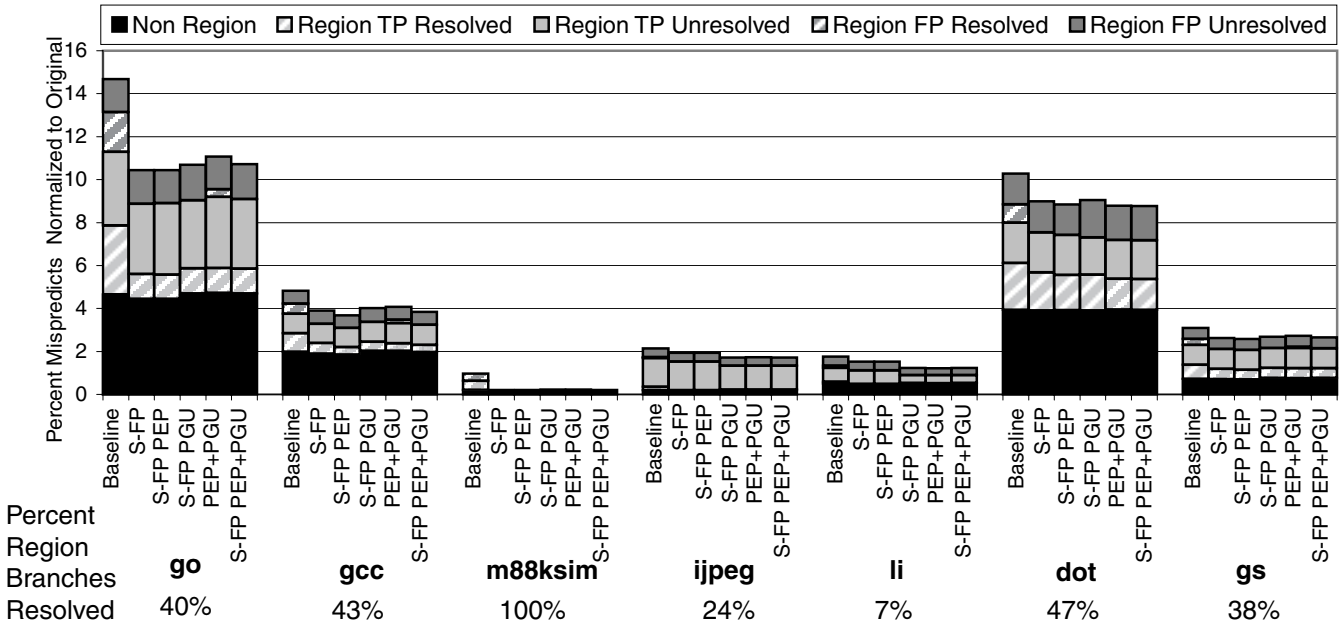


Figure 8: Change in misprediction rate, normalized to the number of predictor accesses in the original unpredicated code. Columns 1 and 5 of each group are not optimized with Squash-FP, the rest show the benefit gained by removing resolved false path predictions. The percent of region branches whose guarding predicate is resolved by branch fetch is shown below each benchmark. PGU can still improve prediction rate when few branch guarding predicates are resolved.

FP architectures would most like take more than one cycle for their prediction. If this was the case, they would then act as a "corrective" predictor correcting the single cycle predictor's prediction if it disagreed with it. Actually, the correction could be provided at any stage in the pipeline. For example, as soon as a branch's guarding predicate is known to be false, and if the branch was predicted as taken, we can redirect the branch down the not-taken path even before the branch has executed.

The critical path of the PEP and Squash-FP architectures requires a two table lookup to perform a 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 the guarding predicate for the fetched branch along with its 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. Similarly, the Squash-FP architecture looks up the guarding predicate of branch in the BTB, then queries the predicate register file and pipeline structures to determine if the guarding predicate has both a false value and has no remaining outstanding defines. 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 global history-based branch prediction architectures.

7.8 Region Formation to Study Branch Prediction

The goal of this work was to develop new branch prediction architectures that improve the accuracy of predicting branches in the presence of predication. In particular we concentrate on improving branches guarded by a predicate register. This includes conditional, unconditional jumps, procedure calls, and returns when they are left inside of a predicated region. To this end, we follow a region formation technique focussed on the elimination of branch mispredictions.

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. Figure 9 shows the percent increase in instructions fetched due to our region formation. This is the percent increase of falsely guarded instructions that are fetched and passed through the pipeline. The percent increase of falsely guarded instructions for `m88ksim`, `ijpeg`, and `dot` is small. From our experience with these programs, they should result in reasonable speedups using the predicated regions we formed. This is because of the significant decrease in branch miss rates shown for these programs in Figure 6 when using these predicated regions. The regions we examine for `go`, `gcc` and `gs` need to be more conservative to reduce the number of false path instructions, or we need to reduce the number of regions predicated.

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,

go	gcc	m88ksim	ijpeg	li	dot	gs	average
94%	50%	20%	11%	24%	10%	77%	41%

Figure 9: Increase in dynamic instructions fetched for our hard-to-predict predicate region formation and scheduling techniques.

we focused region formation on the hard-to-predict branches. To do this we found that we had to allow unbiased, though originally predictable, branches to reside in predicated regions. On average, predicate region formation reduced the branch mispredict rate 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 defines is no longer available in the global history register.

The first technique we examine is to concentrate on branches whose guarding predicate is false using the Squash-FP Filter. Squash-FP achieves 100% prediction accuracy for region branches whose guarding predicate definitions have resolved by the time the branch is fetched. These falsely guarded branches should always predict "not-taken". We show that, even alone, the Squash-FP method of utilizing predicate define information achieves a sizable reduction in branch mispredictions (ranging from 0.5% to 4.3%). This method is arguably the simplest 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 important improvements.

The second approach we examine is a 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 global history register using our deterministic update table. We update the global history register in a dynamically reproducible manner with a deterministic update table triggering predicate define updates of the global history register.

The benefits of a deterministic update branch architecture have potential outside the application for which it was used in this work. Previously, branch predictors have only incorporated instruction information available in fetch (traditionally the direction of other branches). Our deterministic update architecture provides a deterministic mechanism for incorporating information from execution into the branch predictor. This can include instructions immediately before the branch is fetched, from any type of instruction, and any type of information (e.g., if a load missed or hit in the cache).

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] D. I. August, D. A. Connors, J. C. Gyllenhaal, and W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *The 3rd Intl. Symp. on High-Performance Computer Architecture*, pages 84–93, 1997.

[2] 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 *Proc. of the 25th Intl. Symp. on Computer Architecture*, July 1998.

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

[4] 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.

[5] Y. Choi, A. Knies, L. Gerke, and T.F. Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *Proc. of the 34th Annual Intl. Symp. on Microarchitecture*, December 2001.

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

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

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

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

[10] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proc. of the 18th Annual Intl. 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 *Proc. 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 *Proc. of the 25th Annual Intl. Symp. on Microarchitecture*, pages 45–54, December 1992.

[13] S. A. Mahlke and B. K. Natarajan. Compiler synthesized dynamic branch prediction. In *Proc. of the 29th Annual Intl. Symp. 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] 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.

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

[17] G. S. Tyson. The effects of predicated execution on branch prediction. In *Proc. of the 27th Annual Intl. Symp. on Microarchitecture*, pages 196–206, November 30–December 2, 1994.