Branch Prediction in Multi-Threaded Processors

Jayanth Gummaraju
ECE Department
University of Maryland
College Park, MD 20742
jayanth@eng.umd.edu

Manoj Franklin
ECE Department and UMIACS
University of Maryland
College Park, MD 20742
manoj@eng.umd.edu

Abstract

There has been a growing interest in the use of multi-threading to speed up the execution of a single program. This paper highlights the problems involved in performing accurate branch predictions in single-program multi-threaded (SPMT) processors, where branches are predicted out of program order, and microarchitectural aspects affect branch history. In particular, it shows that the recorded branch history could become insufficient, discontinuous, outdated, scrambled, or inaccurate, depending on the specifics of the microarchitecture. Measurements obtained with a multiscalar simulator show that the affects on the recorded branch history cause the branch misprediction ratios to increase from an average of 6.5% to an average of 41.0% for global branch history-based schemes such as gshare, and from an average of 6.7% to an average of 11.5% for per-address history-based schemes such as Pag. The paper also investigates techniques for overcoming the adverse effects of the microarchitecture's impact on branch history. These techniques rely on extrapolation of outdated history so as to make it up-to-date, and correlation with thread-level information so as to partition scrambled history into a set of unscrambled history parts. Experimental evaluation of the proposed techniques on a multiscalar simulator indicates that these techniques, especially a hybrid of extrapolation and correlation, can substantially lower the branch misprediction ratios.

1 Introduction

There has been a growing interest in the use of multi-threading to speed up the execution of a single program [1] [2] [6] [9] [11] [12]. The compiler or the hardware extracts threads from a sequential program, and the hardware executes multiple threads in parallel, most likely with the help of multiple processing elements (PEs). Whereas a single-threaded processor can only extract parallelism from a group of adjacent instructions that fit in a single dynamic scheduler, a single-program multi-threaded (SPMT) processor can extract parallelism from multiple, non-adjacent, regions of a program's execution.

Previous studies have found that to obtain good performance in an SPMT processor, it is important to perform branch prediction within each thread, and speculatively forward data values to subsequent instructions [5]. That is, when the hardware encounters a conditional branch in a thread, it must predict an outcome for that branch, and perform speculative execution along the predicted path, instead of waiting for the branch to be resolved. However, accurate prediction of branches becomes difficult in an SPMT environment, because of the following reasons.

When multiple threads of a sequential program are fetched and executed in parallel, branch instructions belonging to these threads are often fetched (and predicted) in an order different from the dynamic program order. If all of the active threads share a common branch predictor, then the latest branch history may not be available for predicting the outcome of these intra-thread branches. What is worse, the extent to which the available history is outdated may vary for different instances of a branch. Pattern-based prediction schemes such as *gshare* [7] and *Pag* [13], which tend to perform the best in a single-threaded environment, are not likely to perform well in such situations.

If, on the other hand, each active thread has a private branch predictor, then the history maintained by each predictor only includes slices from the overall program-order branch history. This is likely to result in lower prediction accuracy, unless the threads are large enough to build a substantial amount of self-history.

It is important to obtain high branch prediction accuracies in multi-threaded processors. Otherwise, many of the intra-thread control speculations will be incorrect, and the advantages gained by multi-threading are lost. This paper investigates branch prediction in SPMT processors. It analyzes different scenarios that lead to incorrect predictions, and investigates techniques to perform accurate branch prediction in SPMT processors. These techniques attempt to either reconstruct the correct branch history pattern (using "extrapolation" techniques) or make use of correlation with thread-level control flow predictions.

The rest of this paper is organized as follows. Section 2 illustrates how multi-threading impacts branch history, when using prediction schemes that were originally developed for a single-threaded environment (such as *gshare* and *Pag*). Section 3 presents an experimental evaluation of existing predictors in an SPMT environment. Section 4 discusses techniques to overcome the impact of multi-threading on branch history, without significant increase in hardware cost. Section 5 presents an experimental evaluation of the proposed schemes using the multiscalar architecture. Section 6 concludes the paper with a summary.

2 Impact of Multi-Threading on Branch Prediction

2.1 Background and Motivation

During the past decade, a number of dynamic branch predictors have been proposed for single-threaded processors, the most effective ones being pattern-based schemes such as *gshare* [7] and *Pag* [13]. The basic concept behind these predictors is to record the recent branch outcomes, and to make predictions based on the recorded history. The manner in which the history is recorded differs from one prediction scheme to another. Saturating counter-based schemes record the history using a counter that keeps a tally of the number of recent taken outcomes minus the number of recent untaken outcomes. Pattern-based schemes record the recent outcomes as a pattern. Predictors also differ in how they use the recorded history to make a prediction. Single-level predictors have a fixed prediction value associated with each possible history state. Two-level predictors store a second level of history to keep track of the behavior of each first-level history state, and use the second-level history to make predictions.

In conventional single-threaded processors, the branch history is recorded in program order; the recorded history thus depends only on the program being executed and the set of inputs with which the program is being executed (barring the effects of exceptional events such as context switches). Thus, the history available for a particular dynamic branch instance depends only on the program order and the specifics of the predictor, and not on any microarchitectural feature of the processor. In other words, apart from the specifics of the predictor, the microarchitecture does not play any role in deciding the history available for each branch prediction[1].

In an SPMT processor, by contrast, *the multi-threading aspect of the microarchitecture plays a major role in determining the history recorded in the branch predictor.* When multiple threads are executed in parallel, instructions from these threads are generally fetched in an order different from that specified in the program. Naturally, branches from multiple threads are also likely to be fetched out-of-order. Because branch predictions are typically done in the fetch stage of the pipeline, branch predictions are also likely to happen out-of-order. This means that, quite often, the prediction for a branch is performed before several branches preceding it in the dynamic program order have been encountered by the SPMT hardware. The history used by the predictor to perform a prediction for such a branch is not as per program order, and is dependent on the specifics of the microarchitecture. In particular, as we will see shortly, microarchitectural impact can result in 5 types of recorded branch history:

- **Insufficient history:** the recorded history includes very few updates.
- **Discontinuous history:** the recorded history does not include some of the updates.
- **Outdated history:** the recorded history does not contain the latest updates.
- **Scrambled history:** the recorded history is updated in an incorrect order.
- **Inaccurate history:** the recorded history contains erroneous updates.

When the microarchitecture influences branch history in this manner, it is not obvious if the recorded history is capable of predicting future behavior accurately, because of the introduction of several uncertainties. For instance, if one of the threads encounters an i-cache miss, its branch instructions are fetched much later than when there is no i-cache miss.

[1] If a processor updates branch history in the commit stage of the pipeline, then the history available at the fetch stage of the pipeline for predicting a branch may not reflect the most recent history, because there is a time lag between the predictions and the updates. Single-threaded processors can eliminate this microarchitectural dependence by allowing the predictor to perform speculative updates of history, as and when it makes a prediction, with provision to recover whenever a prediction is wrong [3]. This enables the predictor to have correct, up-to-date history for predicting the outcome of branches belonging to the correct path.

2.2 Microarchitectural Issues Impacting Branch History

Let us see how specific microarchitectural issues impact the history recorded by a branch predictor. For this discussion, we use the example depicted in Figure 1.

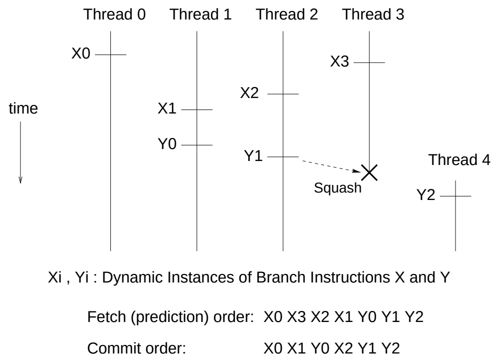

Figure 1: **Example Illustrating the Execution of Multiple Threads Containing Multiple Branch Instances.** There are two static branch instructions, `X` and `Y`. Their dynamic instances are `X0`, `X1`, `X2`, `X3`, and `Y0`, `Y1`, `Y2`. Thread 3 is initiated due to `Y1`'s incorrect prediction; it is squashed when this misprediction is detected.

2.2.1 Thread-Predictor Mapping

An important microarchitectural issue, which affects our entire discussion, is the number of predictors in the multi-threading hardware. In particular, one can think of the following two options:

- Private predictors
- Shared predictor

In the first case, each of the parallelly executed threads uses a separate private branch predictor. That is, the multi-threaded processor has as many predictors as the maximum number of active threads permitted. When a new thread is initiated, it is assigned a private predictor. The residual history present in the predictor can be either reset, or retained for use during the "cold start" period of the thread. In the former case, the history recorded in each private predictor tend to be insufficient, unless the threads are rather long. In the latter case, the history recorded in each private predictor is discontinuous. A motivation for using private predictors, especially for processors geared for large threads, is that they do not use a centralized hardware resource, and therefore permit the predictor hardware to be decentralized. When the thread size is small, the performance is likely to be poor because of *insufficient history* or *discontinuous history*: the history that a branch needs for making a correct prediction is split across multiple predictors.

Alternatively, a single predictor can be shared by all active threads of the processor. The motivation behind such a shared predictor is reduced hardware cost, which permits the use of a complex predictor. When a branch is fetched in any thread, the shared predictor is consulted to obtain a prediction. The exact history recorded in a shared predictor depends on the time at which the history is updated, which we discuss next.

2.2.2 Branch History Update Time

When using a shared predictor, the time at which history update is done has a strong bearing on the recorded history. In particular, there are at least three different times at which the update can be performed:

- branch prediction time (branch fetch time)
- branch resolution time
- branch commit time

If the update is performed at branch prediction time, as in a speculative superscalar processor [3], then the update is speculative in nature. Such an update provides the latest branch history for subsequent predictions. However, unlike in a speculative superscalar processor, in a multi-threaded processor, speculative updates done at prediction time can happen out of program order, resulting in *scrambled history*. Consider the scenario in Figure 1. The correct program order of the instances of branch `X` is `X0 X1 X2`. Their fetch and prediction order, however, is `X0 X3 X2 X1`. If the branch history is speculatively updated after each prediction, then the recorded history is *scrambled*, and the history used for each of the predictions of `X`, except the first one, is *incorrect*. Now, consider the instances of `Y`. Their prediction order, `Y0 Y1 Y2`, is same as the commit order. Therefore, a per-address branch predictor such as *Pag* will see a non-scrambled history while predicting the instances of `Y`, and will be able to predict the outcomes of `Y` correctly if speculative update is performed at prediction time. A global-address predictor such as *gshare* will encounter *scrambled history* for these instances also.

When a branch instruction is squashed due to a misprediction, its history has to be removed from the speculatively updated history so as to keep the history accurate. For instance, in Figure 1, when the incorrect prediction for `Y1` causes `X3` to be squashed, the speculative update done for `X3` has to be removed from the recorded history. In a shared predictor, this is often difficult to do, as the speculative updates happen out of program order.

The second option is to perform the update at branch execution/resolution time. Updation at branch resolution time is similar in many respects to updating at prediction time in that it also happens out of program order. Depending on the execution model, these updates could be speculative or non-speculative.

The third option is to perform the history updates at branch commit time. A motivation for such an update is that the updates happen strictly in program order, assuming that branches are committed in program order. However, as far as predictions are concerned, the updates are delayed, and the available history is *outdated*. For example, in Figure 1, assume that the threads are committed in program order (i.e., `Thread 0`, `Thread 1`, `Thread 2`, `Thread 4`) after all of the branches have been fetched and predicted. Then, the history used for predicting all instances of `X` is the same—the one that was available before any of these threads were initiated! For some programs, the effect of *outdated history* may be more severe than that of *scrambled history*, and vice versa.

It is important to note that the recorded history of a shared per-address predictor (such as *Pag*) is affected only when there are multiple instances of the same branch in multiple simultaneously active threads. The recorded history of a shared global predictor (such as *gshare*) is affected whenever multiple branches are present in multiple simultaneously active threads.

2.2.3 Thread Length

Thread length is an important factor in deciding the history recorded in a predictor. For the sake of this discussion, we can classify threads into two categories, based on their length:

- Short threads
- Long threads

Short threads do not have enough self-history to build, and so are likely to perform poorly with private predictors. Long threads, on the other hand, are likely to develop enough self-history after their "cold start" period, and are likely to perform well with private predictors. For shared predictors, the situation is just the opposite. Branches from two or more long threads are likely to cause more interference, whereas branches from short threads are less likely to cause interference.

2.2.4 Intra-Thread Control Flow and Thread Execution Style

The exact nature of threads and their execution style have a strong bearing on branch history and branch prediction. For instance, if threads are initiated speculatively (as in the multiscalar processor [1] [11] and the superthreaded processor [12]), then some of the active threads may get squashed because of incorrect thread-level control speculation. When using a shared branch predictor, if the updates are done at branch prediction time, then thread-level misprediction requires setting back some of the speculative updates selectively. When updates are not done in program order, it becomes very difficult to do these setbacks correctly.

A related issue is the number of alternate flows of control embedded in a thread. If a thread encompasses many alternate flows of control, then speculatively updated history is more likely to be *scrambled*. On the other extreme, if a thread is allowed to have only a single flow of control, as in trace processors [9], there is no need for intra-thread branch prediction.

Another issue relates to whether parallelly executed threads can have instances of the same branch, which can occur when the threads are iterations of the same loop. If loop iterations are executed in parallel, then per-address predictors such as Pag are affected, because of the likelihood of multiple instances of a branch to be simultaneously present in the processor.

3 Evaluation of Existing Branch Predictors in an SPMT Environment

The previous section identified several microarchitectural issues affecting branch history and branch prediction in a multi-threaded processor. In this section, we evaluate how they impact the performance of existing predictors in an SPMT environment. In particular, we use a simulator for the multiscalar processor [1]. This simulator performs detailed cycle-by-cycle simulation of the multiscalar SPMT processor-memory system. We evaluate both private predictors and shared predictors, with three different predictors in each category (*3-bit saturating counter*, *gshare*, and *Pag*). For the shared predictor cases, for each predictor, we evaluate two history update schemes—update at commit time and update at resolution time. For comparison, we have also included the corresponding prediction accuracies obtained in a single-threaded environment.

For benchmarks, we use one program from the SPEC '92 suite and the remaining from the SPEC '95 integer suite. They are compiled using the MIPS C compiler with the optimization flags distributed in the SPEC benchmark makefiles. Notice that the code was compiled for a single-issue processor. The benchmarks are simulated up to 100 million instructions. The exact results are dependent on the nature of the threads and the execution model, and are likely to change when these are changed. The default parameters are:

- *Nature of Threads:* Each thread is a subgraph of the control flow graph embedded in the static binary, with up to 4 control arcs emanating from it, but is a contiguous section of code from the binary. A thread can have a maximum of 32 instructions.
- *Thread-Level Predictor:* This is a 2-level Pap predictor. Its first-level table has 8K entries. The pattern size is 6, and so the second-level table (PHT) has 64 entries, each of which has four 3-bit up/down saturating counters.
- *Degree of Multi-threading:* All multi-threaded configurations support up to 8 active threads.
- *Pattern History Table (PHT) for Intra-Thread Branch Predictors:* All of the pattern-based intra-thread branch predictors use a pattern size of 8. Thus, the PHT size is 256. Each of the PHT entries is a 3-bit up/down saturating counter.

3.1 Private Predictors

Figure 2 presents the misprediction ratios for the three private predictors. On inspecting the three sets of histograms, a few comments are in order. First, the *3-bit SC* scheme, which is non-pattern-based and does not have very high accuracy in a single-threaded environment, is hardly affected by *discontinuous history.* This is because, conceptually, the history stored by a saturating counter-based predictor is not "exact," but only an "average" of the recent history. Discontinuities in the history do not make a big impact on the "average" history recorded. Second, the performance of the pattern-based *Pag* scheme, which only uses per-address history, is somewhat affected when the thread size is 32. Discontinuous history has a bigger impact on Pag, because the history used by Pag is "exact" in some sense, and not an "average." However, the impact is not as severe as in *gshare*, because the history is on a per-address basis.

In order to see how private predictors perform with large threads, we conducted another study by varying the thread length from 32 to 2K instructions. Because we do not have a multiscalar compiler that can generate arbitrary size threads, this study was done using a trace-based simulator. These results are obtained by simulating 8 predictors for each thread length (L), and by using predictor $\lfloor \frac{I}{L} \rfloor$ mod 8, for the branch that is the I^{th} dynamic instruction in the program. The predictor is updated with the correct outcome immediately after the prediction, as in any trace-based simulation setup. The branch history present in a predictor is not reset when a new thread is initiated, so the history available in each predictor is *discontinuous.* Figure 3 plots the branch misprediction ratios obtained for various thread lengths.

As we can see from Figure 3, when the thread size is increased, private Pags begin to perform better. This is because each thread has many instances of a branch, and once the "cold start" of each thread is over, the discontinuity in the recorded history slowly disappears. The results reported in Figure 3 indicate that none of the private, per-thread predictors is adequate when the thread size is small, and that per-thread Pag predictor works well when the thread size is large. In particular, for all of the programs considered, a thread size of 2K instructions seems adequate for private Pag predictors to provide prediction accuracies close to that obtained in a single-threaded environment.

3.2 Shared Predictors

The next set of results are for shared predictors. The histograms in Figure 4 show how shared *3-bit SC*, *Pag*, and *gshare* predictors perform in an SPMT environment when branch history is updated at resolution time (RT) and at commit time (CT). All of these schemes perform only non-speculative updates.

First, let us analyze the results for the shared *3-bit SC predictor.* A comparison of the first and third histogram bars for each benchmark shows that its prediction accuracy is hardly affected by outdated history. An inspection of the middle bar for each benchmark shows that scrambled and inaccurate histories do have a small impact on the performance of the *3-bit SC* predictor.

Next, consider the histograms for the shared Pag predictor. The most spectacular increase is seen for `espresso`, from 4% to 20%. Other programs also suffer a marked increase in misprediction ratio. For `espresso`, the shared *Pag* predictor is performing worse than the shared *3-bit SC* predictor, for the multi-threaded configuration simulated!

Finally, the results for the shared *gshare* predictor indicate that its misprediction ratio is very high for small thread sizes, somewhat high for larger thread sizes. Thus, *branch prediction schemes that depend on pattern-based history (and which perform the best in single-threaded systems) tend to suffer the most in multi-threaded systems.*

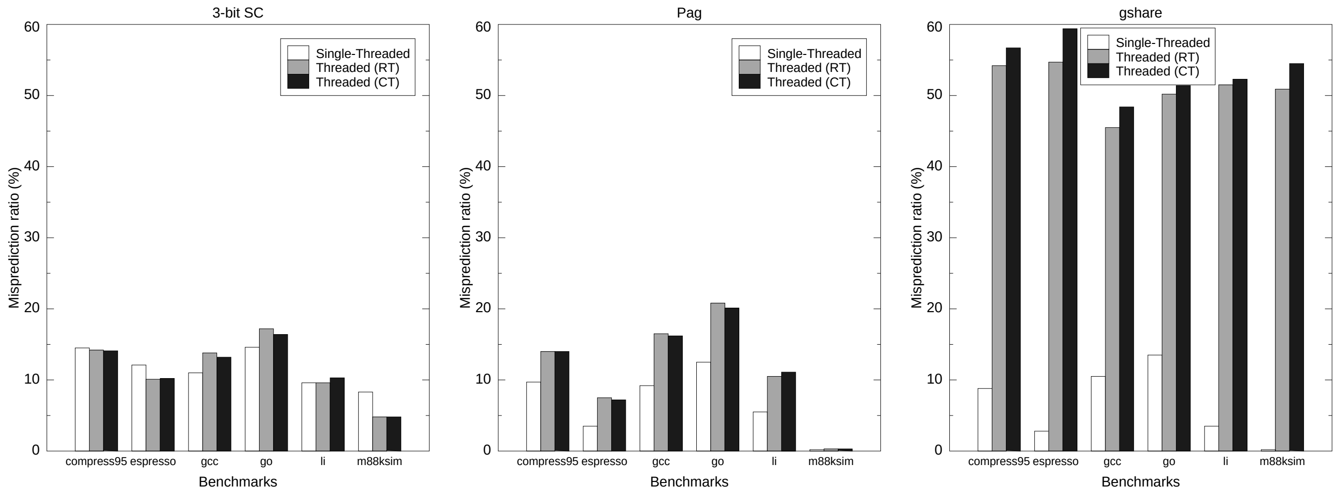

Figure 2: **Branch Misprediction Ratios for Three Different Private Predictors.** The Y-axis plots the branch misprediction ratio as a percentage. Each benchmark has 3 bars in each histogram, corresponding to a single-threaded environment, multi-threaded environment with history update at branch resolution time (RT), and multi-threaded environment with history update at branch commit time (CT).

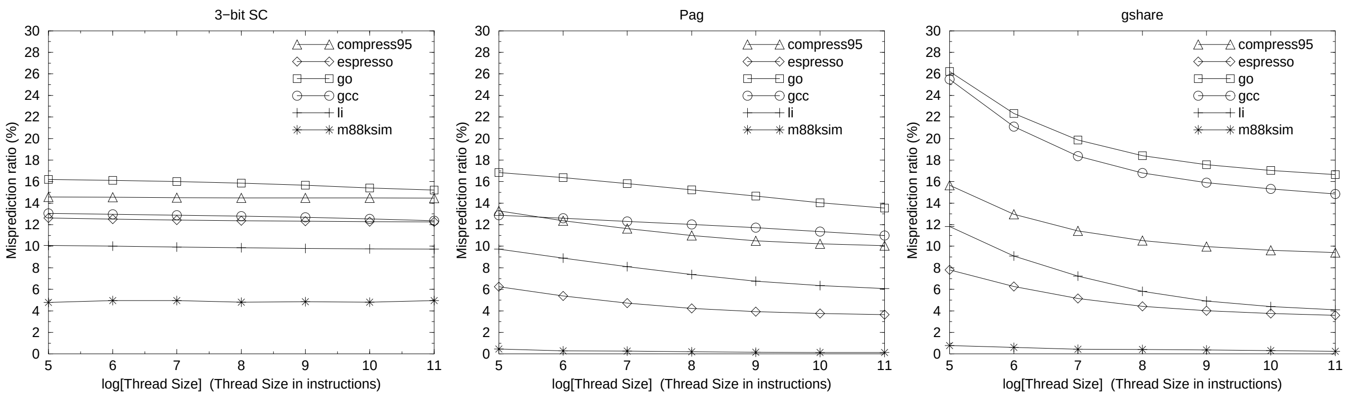

Figure 3: **Branch Misprediction Ratios for Three Different Private Predictors for Varying Thread Lengths.** The X-axis plots, in logarithmic scale, thread lengths ranging from 32 to 2K instructions; thread length of ∞ would correspond to a single-threaded environment. The Y-axis plots the branch misprediction ratio as a percentage.

4 Branch Prediction Techniques for SPMT Systems

The previous sections highlighted the impact of multithreading on branch history and branch prediction accuracy. This section deals with correcting these problems. As mentioned earlier, this paper focuses primarily on SPMT systems that use relatively short threads, that is, thread lengths ranging up to 100 instructions or so. As shown earlier, when the threads are large, private, per-thread, branch predictors tend to perform quite well. When the thread size is small, neither the shared predictor nor the private predictors work well. In the following subsections, new schemes are proposed which are easy to implement, does not require significant extra hardware, and give prediction accuracies close to those obtained in single-threaded systems.

4.1 Extrapolation to Obtain Up-to-Date History

The extrapolation technique attempts to reconstruct the history that would be available, had the processor been updating branch history in program order and in

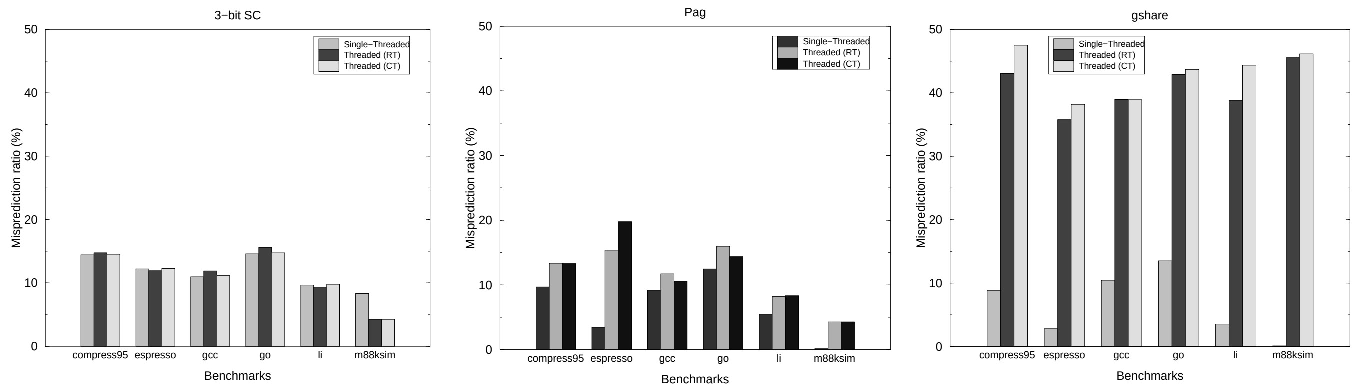

Figure 4: **Branch Misprediction Ratios for Three Different Shared Predictors.** The Y-axis plots the branch misprediction ratio as a percentage. Each benchmark has 3 bars in each histogram, corresponding to a single-threaded environment, multi-threaded environment with history update at branch resolution time (RT), and multi-threaded environment with history update at branch commit time (CT).

a timely manner. Extrapolation is particularly useful for predicting the instances of branches that belong to a loop, and are fetched in program order.

Figure 5 represents a typical implementation of an Extrapolation-based Pag predictor. The first level includes a Branch History Table (BHT) and the second level includes a Pattern History Table (PHT), just like in a typical two-level predictor. The BHT contains two parts, namely `Branch History` and `Speculated Updates`. The former records the last few outcomes (non-speculative updates) of a branch, and the latter stores the subsequent predictions for the pattern present in the corresponding `Branch History`. The PHT contains `Saturating Counters` that keep track of the behavior of the patterns in the BHT. When a branch is fetched, a few bits from its PC are used to index into the BHT. Based on the extrapolation distance, the correct bit from the `Speculated Updates` part is chosen using a MUX. This gives the prediction for the branch. The extrapolation distance is calculated based on the number of active instances of the branch (fetched but yet to be committed) prior to this branch instance. When a branch gets committed, based on the pattern in the BHT and the branch outcome, the PHT is updated. After this, the pattern in the BHT is updated and the next few outcomes are speculated based on the current pattern, and are stored in the `Speculated Updates` part of the BHT. Because the updates are done at commit time, the correct order is maintained.

Note that this is just one of several ways of implementing this predictor. Although extrapolation of outdated history appears to be similar to speculative update at prediction time, there is a difference between

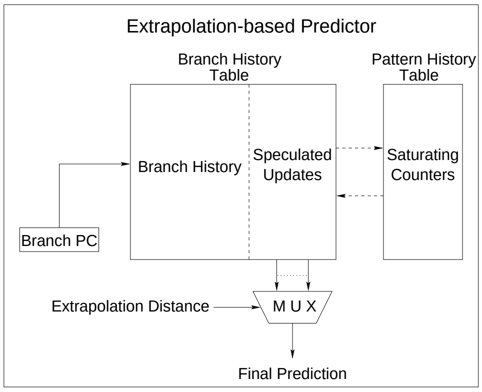

Figure 5: **Block Diagram of Extrapolation-based Shared Per-Address Predictor.**

the two. This can be explained with the help of the example in Figure 1. The correct program order of the instances of branch `X` in the example is `X0 X1 X2`. Their fetch and prediction order, however, is `X0 X3 X2 X1`. If the branch history is speculatively updated after each prediction, then the recorded history is *scrambled*, and the history used for each of the predictions of `X`, except the first one, is *incorrect*. On the other hand, if the history is updated at commit time and extrapolation is performed at prediction time, the prediction for `X1` would also use correct history, provided the relative seniority of the threads is considered when calculating the extrapolation distance.

4.2 Correlation with Thread-Level Information to Obtain Accurate History

We saw that the extrapolation scheme tends to fail when multiple instances of a branch are not fetched in program order. In such situations, instead of attempting an erroneous re-construction of the history, it may be more appropriate to isolate different kinds of contexts. A good way to do that is to correlate the history with thread-level information[2]. Correlation with thread-level predictions attempts to add that touch of global branch behavior which a shared Pag predictor lacks. Consider a scenario in which history update is done at commit time, and the history is currently up-to-date. If four instances of a branch are then fetched (and predicted) before performing any further updates, then only the first one among them uses up-to-date history. If, on the other hand, the history is partitioned based on the thread context and recorded by multiple predictors (one for each context), then all four instances get to use up-to-date history, provided they belong to different contexts.

Correlation also helps when history updates are done at branch resolution time or branch prediction time. Consider the program order branch history pattern `110110110....` Assume that out-of-order updates cause the corresponding branch history pattern to be recorded instead as `111100110....` (scrambled history). Assume also that the corresponding thread has 4 possible control outcomes A, B, C, D, and that whenever the thread-level control outcome was `A`, the branch outcome was 0. If the branch history is correlated with the thread-level control speculation value, then the scrambled pattern of `111100110....` gets split into 4 unscrambled and accurate pattern streams as follows: `1111...`, `1111...`, `0000...`, and `1111....`

Figure 6 shows the block diagram of a correlation based predictor. The predictors (`Predictor 1, ... n`) are indexed based on the branch PC value, and each predictor gives its prediction. Based on the higher (thread) level prediction, an appropriate prediction is chosen using a MUX. The higher level information thus guides the lower level in making predictions. When the branch is committed, the predictor is also updated. Note that as in the extrapolation-based predictor, the correct order of updates is maintained.

[2]Correlation has been used in the past [8] to improve branch prediction accuracy by correlating the history of a branch with the outcomes of the previous two dynamic branches in the program.

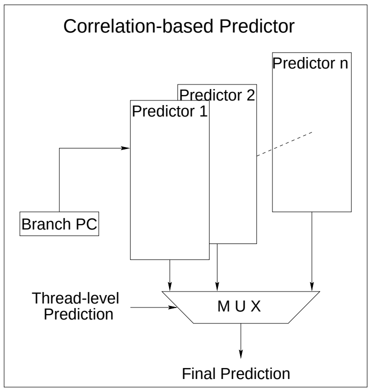

Figure 6: **Block Diagram of Correlation-based Shared Per-Address Predictor.**

4.3 Hybrid of Extrapolation and Correlation

The techniques of extrapolation and correlation discussed above are complementary. Whereas extrapolation attempts to make outdated history up-to-date, correlation steers a prediction request to a specific predictor based on the thread context to which the branch belongs. It is possible for a specific predictor history to be outdated, owing to uncommitted branches belonging to that context. The prediction accuracy can be improved in this situation by employing both extrapolation and correlation together, in a hybrid predictor.

Figure 7 shows a block diagram of a hybrid predictor that employs both extrapolation and correlation. The organization is similar to that of the correlation-based predictor given in Figure 6, except that it uses an extrapolation-based predictor as each of the component predictors.

5 Experimental Evaluation

This section presents an experimental evaluation of the techniques discussed in Section 4 for overcoming the impact of multi-threading on branch history. The experiments in this section are conducted using the same multiscalar simulator that was used to obtain the results in Section 3.

5.1 Prediction Schemes Evaluated

- Shared Extrapolation-based Pag Predictor, with a direct-mapped BHT of 4K entries.
- Shared Correlation-based Pag Predictor For this technique, we simulate two schemes:

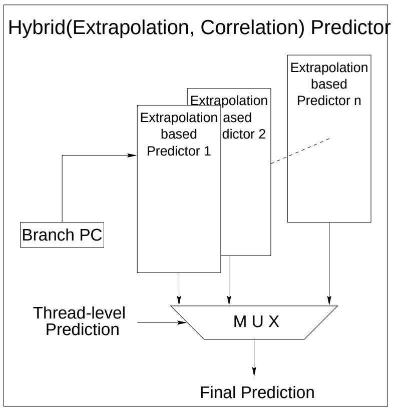

Figure 7: **Block Diagram of Hybrid (Extrapolation, Correlation) Shared Per-Address Predictor.**

- Correlation based on predicted successor of current thread — *Correlation (Target)* — with four 1024-entry direct-mapped BHTs.
- Correlation based on predicted successor's position in static program relative to current thread — *Correlation (Approx)* — with three 1024-entry direct-mapped BHTs.

- Shared Hybrid (Extrapolation, Correlation (Target)) Pag Predictor, with four 1024-entry direct-mapped BHTs.

Note that the total number of entries in the first level table(s) are the same for the single-threaded predictor, the extrapolation-based predictor, the correlation (target)-based predictor, and the hybrid predictor.

5.2 Experimental Results

Figure 8 presents the branch misprediction ratios obtained for different prediction schemes. The Y-axis represents the misprediction ratios. Each benchmark has 6 bars, corresponding to shared Pag in an SPMT environment (update at commit time), Extrapolation-based Pag, Correlation (approx)-based Pag, Correlation (target)-based Pag, Hybrid (extrapolation, correlation) Pag, and Pag in a single-threaded environment.

The experimental results indicate that both extrapolation-based and correlation-based schemes reduce the misprediction ratio by a considerable extent for most benchmarks. An exception is `m88ksim` for which the extrapolation scheme increases the misprediction ratio substantially. Extrapolation helps some branches to a large extent. At the same time it abets a number of other branches. The reason is that the

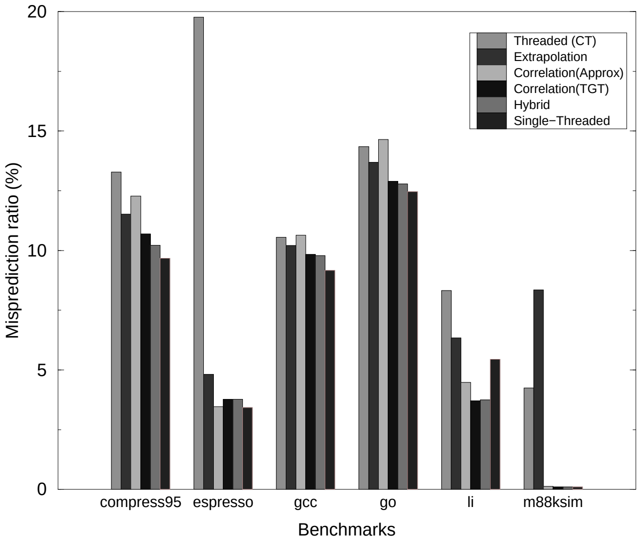

Figure 8: **Branch Misprediction Ratios for Various Prediction Schemes.**

premise under which extrapolation works is that multiple instances of a branch are fetched in program order. As long as this is met, extrapolation works well.

Measurements done for the correlation schemes (not reported here) showed that the number of active instances of a particular branch in each context is substantially less than the number of active instances of the branch in a non-correlation-based scheme. This confirms our hypothesis that the correlation-based scheme steers multiple instances of a branch to different predictors, thereby reducing the probability for scrambled or outdated history. An inspection of the third and fourth bars for each benchmark shows that the correlation (approx) scheme is a good approximation for the correlation (target) scheme for `espresso`, `li`, and `m88ksim`. Correlation (approx) is especially useful when threads can have a large number of potential successors.

Next, consider the results for the hybrid (extrapolation, correlation) predictor. In `espresso` and `li`, the misprediction ratio is reduced by more than 50%. In all the benchmarks except `li`, the misprediction ratio is within 1% of what is obtained in the single-threaded environment. In `li`, it even outperforms the single-threaded case. This is due to the following reasons: (i) for `li` extrapolation by itself is able to provide for the multi-threaded case the same prediction accuracy as that of the single-threaded case, (ii) correlation adds a global component and brings the prediction accuracy closer to that of gshare, which happens to be higher than that of Pag for `li`.

6 Summary and Conclusions

Several researchers and microprocessor vendors are working on single-program multi-threaded (SPMT) processors, to harness additional levels of parallelism. When multiple threads are fetched and executed in parallel, branches belonging to these threads are very likely to be fetched out of order. This impacts the branch history recorded in branch predictors, and affects their capability to accurately predict branch outcomes. With the help of a simulation study, we showed that an existing per-address branch prediction scheme can suffer an average of 11.5% mispredictions, and an existing global-address branch prediction scheme can suffer an average of 41% mispredictions.

We investigated techniques for reducing the number of branch mispredictions in SPMT processors. The first technique performs history updates at commit time (so as to obtain a non-scrambled, but outdated history), and extrapolates the recorded history to obtain up-to-date history. The second technique uses correlation with thread-level information to create different contexts for multiple instances of a branch, so that the recorded history is less scrambled and less out-of-date. The third technique combines extrapolation and correlation to form a hybrid predictor.

These three schemes were evaluated using simulations done with a multiscalar simulator. The results of our experiments show that the hybrid scheme is able to bring down the misprediction ratio to that in a single-threaded processor, for all of our benchmark programs.

The performance of a particular prediction technique is strongly dependent on the kind of threads that are being used. If the threads are such that most of the time the branches are fetched in program order, then extrapolation works well. Updation at commit time seems to be an essentiality, especially when there are a large number of squashes either in the fetch stage or in the execute stage.

Acknowledgements

This work was supported by the US National Science Foundation (NSF) through a CAREER Award (MIP 9702569) and a research grant (CCR 9711566), and by a Graduate Fellowship from the University of Maryland, College Park. We thank the reviewers and Bruce Jacob for their helpful comments. Thanks are also due to Luo Kun, Aneesh Aggarwal, and all the other members of the System and Computer Architecture Laboratory for their constant support and encouragement.

References

[1] M. Franklin, "The Multiscalar Architecture," *Ph.D. Thesis, Technical Report TR 1196*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[2] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," *IEEE MICRO*, March-April 2000.

[3] E. Hao, P-Y. Chang, and Y. N. Patt, "The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited," *Proc. 27th Int'l Symp. on Microarchitecture (MICRO-27)*, 1994.

[4] Q. Jacobson, S. Bennett, N. Sharma and J.E. Smith, "Control Flow Speculation in Multiscalar Processors," *Proc. 3rd Int'l Symp. on High Performance Computer Architecture (HPCA-3)*, 1997.

[5] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," *Proc. 19th 22nd Annual Int'l Symp. on Computer Architecture*, 1992.

[6] Pedro Marcuello and Antonio Gonzlez, "Exploiting Speculative Thread-Level Parallelism on a SMT Processor," *Proc. Int'l Conf. on High Performance Computing and Networking*, 1999.

[7] C. McFarling, "Combining Branch Predictors," *WRL Technical Note TN-36*, June 1993.

[8] S-T. Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 76-84, 1992.

[9] E. Rotenberg, Q. Jacobson, Y. Sazeides, J. E. Smith, "Trace Processors," *Proc. 30th Int'l Symp. on Microarchitecture (MICRO-30)*, 1997.

[10] A. Seznec, S. Jourdan, P. Sainrat, P. Michaud, "Multiple-Block Ahead Branch Predictors," *Proc. 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, 1996.

[11] G. S. Sohi, T. N. Vijaykumar, and S. Breach, "Multiscalar Processors," *Proc. 22nd Int'l Symp. on Computer Architecture*, 1995.

[12] J-Y. Tsai and P-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation," *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT '96)*, 1996.

[13] T-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proc. 19th Int'l Symp. on Computer Architecture*, pp. 124-134, 1992.