Accurate Branch Prediction for Short Threads

Bumyong Choi   Leo Porter   Dean M. Tullsen

University of California, San Diego
La Jolla, California 92093

Abstract

Multi-core processors, with low communication costs and high availability of execution cores, will increase the use of execution and compilation models that use short threads to expose parallelism. Current branch predictors seek to incorporate large amounts of control flow history to maximize accuracy. However, when that history is absent the predictor fails to work as intended. Thus, modern predictors are almost useless for threads below a certain length.

Using a Speculative Multithreaded (SpMT) architecture as an example of a system which generates shorter threads, this work examines techniques to improve branch prediction accuracy when a new thread begins to execute on a different core. This paper proposes a minor change to the branch predictor that gives virtually the same performance on short threads as an idealized predictor that incorporates unknowable pre-history of a spawned speculative thread. At the same time, strong performance on long threads is preserved. The proposed technique sets the global history register of the spawned thread to the initial value of the program counter. This novel and simple design reduces branch mispredicts by 29% and provides as much as a 13% IPC improvement on selected SPEC2000 benchmarks.

Categories and Subject Descriptors C.1.2 [*Processor Architectures*]: Multiprocessors

General Terms Design, Performance

Keywords Chip Multiprocessors, Branch Prediction

1. Introduction

As general purpose processors become increasingly multi-core, we will see the average occupancy time of a single thread on a single core driven down to new levels. This will be a result of several factors. The desire to make better use of the available cores will lead to greater use of architectural and compilation techniques that employ short threads to expose parallelism. Lowered communication latencies, as found between cores on a multi-core architecture, will significantly reduce the cost of migrating a thread between cores, thereby increasing the desirability of migration. As a result, execution models that rely on, or can exploit, shorter threads will become increasingly attractive as the cost of starting a new thread or moving an existing thread is reduced.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'08 March 1–5, Seattle, Washington, USA.

Copyright © 2008 ACM 978-1-59593-958-6/08/03. . . $5.00

This research demonstrates that a significant component of the startup cost of very short threads is the cost of ramping up the branch predictor – and the primary startup cost is not the prediction tables themselves, but rather the global branch history typically used to index those tables. This is because the tables of saturating counters will have the chance to learn from previous instantiations of similar threads, but only if the indexing function is repeatable. However, if the global history register contains no thread-specific information, or worse, contains noise, the predictor will neither be predicting correctly, nor learning usefully, until the history register has filled completely with branch history that corresponds to the new thread of execution. The paper examines this phenomenon in the context of a speculative multithreading architecture that frequently spawns new theads, each with an unknowable branch history at the point of spawning.

Consider an aggressive predictor that has 20-25 bits of global history in the global history register (GHR). For many such predictors, even if the GHR has just a few bits of random noise (GHR bits not relevant to the current execution), the indexed predictor locations will be unlikely to correspond to locations indexed by the previous or next instantiation of the thread. Thus, for any thread that executes less than 50-75 branches (i.e., several hundred instructions), we'd expect the overall branch predict rate to be extremely low. As a point of reference, Marcuello and Gonzalez [23] report a speculative multithreading architecture with an average thread length of 32 instructions, and Zilles and Sohi [52] report an average helper thread length of 34 instructions. In the Transactional Coherency and Consistency paper [6] that investigates the characteristics of common transactions, they report that on 3 out of 6 speculatively parallelized benchmarks the average lengths of their compiler-generated threads (transactions) were quite short – 129, 244, and 521 instructions, respectively.

This paper presents very small architectural changes that significantly improve the processor's ability to predict branches in short threads, while not compromising in any way the ability to predict for longer threads after the GHR is warmed up. Our solution ensures we have repeatable state in the GHR, yet still allows the predictor to be able to distinguish between threads that are not expected to have the same startup behavior. We examine a number of potential mechanisms, and find that a solution which simply uses program counter bits (from the new thread's starting PC) to initialize the GHR performs as well or better than more complex and speculative mechanisms. Our particular focus is on a speculative multithreading architecture running on a multi-core platform. However, the techniques we demonstrate will also apply to helper threads and compiler parallelization techniques that create large numbers of short threads. It will equally apply to all of these architectures utilizing a multithreaded core. In this work, we also present a simple solution to the case where short threads are the result of excessive core migration.

This paper is organized as follows. Section 2 motivates our research by introducing background work that cause short threads

to be more prominent and their implication for branch prediction. Section 3 discusses related work. Section 4 briefly details our speculative multithreading architectural model. Section 5 describes the simulation framework. Section 6 presents various policies to generate useful global history for a newly spawned thread. Section 7 reports our experimental results and Section 8 concludes.

2. Background — The Problem with Short Threads

There are several architectural factors that will continue to drive down the average stay of a thread on a core, all compelled by the fact that multi-core architectures will (1) reduce the communication and memory-based startup costs for thread initiation and (2) increase the availability of extra cores that we will want to employ for performance gains, energy savings, etc.

Specific examples include speculative multithreading [40, 7, 28, 30, 32, 43, 45, 47], or thread-level speculation, which executes multiple not-necessarily-independent short threads in parallel, and detects unpredicted dependence violations – squashing those threads where parallelism was not found. Helper threads [4, 8, 52, 31, 50, 51] execute short threads in support of the original thread of execution, often for cache prefetching. In some cases, a helper thread only computes the address of a single or a few loads, in others it seeks to capture all the iterations of a loop instance.

Chapparo *et al.* [3] propose core-hopping to eliminate or react to thermal emergencies. Thus, the lifetime of a thread on a particular core will be a fraction of the actual thread lifetime. Similar effects are seen in the heterogeneous multi-core proposals [20, 19], where heavy sampling is used to find the best mapping of threads to cores (resulting in frequent switching of threads between cores), or in Annavaram, *et al.* [2] where new threads are initiated at each new instance of parallelism.

In striving to exploit the ever-increasing availability of cores, compilers will employ techniques that create short threads to extract parallelism [15].

All of these techniques, will be sensitive to the startup costs of a thread. By reducing those startup costs, we can increase the viability of these architectures, and possibly enable new architectural and execution paradigms. Clearly, a huge barrier to the use of execution models with data-flow characteristics [10, 33] on a conventional architecture is the thread creation overhead. Reducing this thread initiation overhead is the goal of this research.

We show that the key to quick branch predictor startup is managing the global history register. The global history register was introduced by Yeh and Patt [49] as a special case of their two-level adaptive branch predictor, with the pattern history table collapsed to a single global entry. McFarling [26] described more explicitly the benefits of the GHR to exploit the correlations between different static branches, and demonstrated its usefulness in his *gselect* and *gshare* predictors. Since, nearly every important predictor has used the GHR in some way. It should be pointed out that the two-level predictor's pattern history table creates an even bigger problem, as we lose the context of every entry in the table when we change cores. Our dependence on the GHR has only grown over time. The 21264 (EV6) combining predictor uses a 12-bit GHR to index both the global prediction table and the choice predictor, and the other half of the predictor uses the aforementioned two-level predictor. The EV8 predictor [36], based on the 2Bc-gskew predictor [37], indexes three tables with GHR bits and uses 21 total bits.

Perceptron predictors [17] have been proposed that use as much as 36 bits of global history. Rather than using two-bit saturating counters, these predictors use a simple neural network indexed by PC and trained to individual bits of the GHR by branch outcomes. These predictors present an interesting option in this work. The Perceptron predictor is more resistant to noise than other GHR-based predictors. Thus, it is no longer necessary for the entire GHR to be filled with meaningful branch outcomes for the predictor to start performing well. However, due to their reliance on larger GHRs, it still requires a significant number of branches before the useful data in the predictor begins to dominate the noise. We examine the effectiveness of our techniques on the perceptron predictor in Section 7.

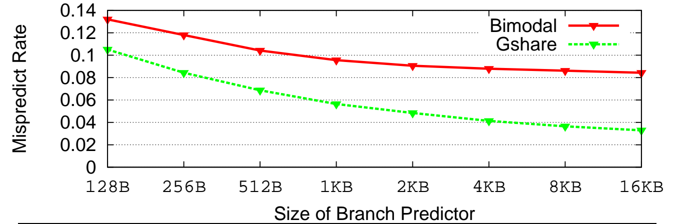


Figure 1. Comparison of a PC-only bimodal predictor and Gshare, for the Spec2K benchmarks.

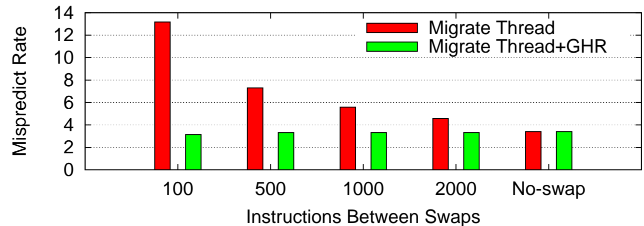


Figure 2. Average mispredict rate across SPEC2K benchmarks when a migration between cores is forced every N instructions (Predictor: 2Bc-gskew).

One alternative solution to our problem would be to simply use the best predictor we could find that did not use the global history register, or any other stored branch pattern; unfortunately, that limits our choices excessively, to essentially a bimodal [39] branch predictor (a simple BTB based predictor or next-Icache-line predictor, in the best case, would match the bimodal predictor). The bimodal is simply a table of saturating counters indexed by PC. In Figure 1, we see that compared to a gshare predictor [26] on the SPEC2K benchmarks, the bimodal predictor gives up way too much performance (both at each point, and asymptotically) to be considered seriously.

Figure 2 illustrates the problem with branch prediction in the presence of short thread occupancy time on a core. In this experiment, we take a single thread and force it to migrate round-robin between four cores, moving every n instructions. We see that (at least for threads of 2000 instructions or smaller) the branch mispredict rate increases, becoming quite significant as the threads get below 500 instructions. In the second bar, where we also migrate the GHR, we see that the mispredict rate is almost completely unaffected. This confirms our thesis that the problem is with the GHR value, not the data in the tables themselves. As long as the GHR is correct, each predictor is able to warm up to the program's branch behavior separately.

However, if we do not migrate a new GHR value, the GHR's global history on a new core is the residual information from the last thread executed on that core. The residual data in the GHR typically does not provide any relevant information with

regard to the execution of the new thread (unless the sequence of executed threads is highly repeatable). The misleading GHR value will be used in the indexing function to produce branch predictions. In addition, the counters associated with those indices will be polluted.

This result presents the first contribution of this paper – that when short threads are caused by excessive core migrations (e.g., the core hopping and heterogeneous swapping examples above), the problem is easily and completely solved by just migrating the GHR with the thread.

However, this does not help in the case of helper threads or compiler-generated short-thread parallelism, because there is no obvious GHR to migrate. Similarly, for speculative multithreading, there is a correct GHR that corresponds to the single-thread execution point of the new thread's starting PC, but that GHR is not known at the time the thread is spawned. Thus, in these scenarios, the thread would likely start with the residual state left in the GHR by the last thread executed, and performance would again resemble the dark bar in Figure 2.

This paper focuses in particular on the Speculative Multithreading (SpMT) problem, but the solution should be equally applicable for the other execution models that incur frequent thread spawns. We will examine several options for introducing (artificial) state into the GHR. Our baseline is the default architecture that maintains whatever bits were in the GHR when the new thread begins execution. At the opposite end of the spectrum, we will examine several ways to predict the value the GHR will have when the spawning thread arrives at the CQIP (the starting point of the spawned thread). Alternatively, we will look at solutions that simply strive to set a consistent starting point for each thread, either by clearing the GHR, or setting it to a thread-specific value.

3. Other Related Work

Most proposed branch predictors of the last 15 years have included some kind of branch pattern history, and usually a global history register. In addition to those predictors described in the previous section, it would also include the bi-mode predictor [39], the agree predictor [41], the YAGS predictor [11], and several flavors of the Perceptron predictor [17, 16], already mentioned.

More recently proposed branch predictors show that even longer histories can provide improved performance. The O-GEHL [34] predictor uses histories from 100-200 bits long, the L-TAGE [35] predictor uses a history length of 4 to 640 bits, and PPM [5] based predictors also rely on very long history lengths to determine the closest matching previous pattern. Although O-GEHL and L-TAGE have the ability to adapt to using shorter histories, longer histories are still necessary, for some benchmarks, for high accuracy.

We examine branch predictors for short threads in the context of a speculative multithreading architecture. A wide variety of SpMT models have been proposed since the original Multiscalar architecture [40]. Those proposed include models that work best when threads are largely independent [44, 7] and those that work even in the presence of dependences [24, 32]. It includes those that squash violating threads completely, and those with the ability to incorporate individual instruction results of a violating thread [1, 21]. Some models rely primarily on hardware prediction of live-ins [24] and others use software to compute register and memory live-ins [32]. Recently, it has been shown that an architecture that supports transactional memory can do some amount of thread-level speculation [6, 13, 48].

Gummaraju and Franklin address branch prediction on the Multiscalar [12] speculative multithreading core. They see branch mispredict rates rise to 41% in that architecture. However, their solution focuses on a two-level local branch predictor (no GHR, but lots of per-address branch history to get corrupted) which is shared among the threads. Thus, much of their focus is on attempting to try to reconstruct a particular branch's history pattern in the face of branches fetching and committing out of order. They are able to gain a significant amount of the performance back for their architecture. Unfortunately, those techniques do not translate either to our predictors (with heavy reliance on the GHR) or to our architecture (CMP with separate predictors and decoupled history).

Several papers examine branch prediction on simultaneous multithreading processors [14, 29]. In general, branch prediction was not shown to be a significant problem for that architecture, despite sharing of a single physical branch predictor. However, none of those studies examined an execution model with short threads or frequent movement of thread control flow across thread contexts. Because we show that the problem is not the lack of predictor table state, but rather the GHR, an SMT processor would be just as vulnerable to these problems as a multi-core processor.

The effectiveness of the solutions examined in this paper will be somewhat sensitive to the details of the speculative multithreading architecture we use to test them, so we describe our architecture in detail in the next section.

4. Baseline Architecture

Our underlying architecture implements a form of speculative multithreading. A wide variety of SpMT execution models have been proposed. This section details the particular architecture we simulate to evaluate our different branch prediction optimizations.

4.1 Speculative Multi-threading

To explore the performance improvement of branch prediction techniques on short threads, we employ a SpMT architecture which causes frequent thread startup due to the spawning of *speculative threads*. A SpMT processor improves single thread performance by splitting the execution into several speculative threads which are executed on different cores. A speculative thread begins execution with a speculative architectural state, possibly including predicted register and memory values. By running the speculative thread(s) in parallel with the non-speculative thread, the SpMT processor exploits intra-thread parallelism.

Using the terminology from [32], a speculative thread is started when the first address from a spawning pair is fetched by the processor. A spawning pair includes two instructions that are referred to as the Spawning Point (SP) and the Control Quasi-Independent Point (CQIP), respectively. The SP is an instruction that triggers the processor to create a new speculative thread. The CQIP is the first instruction that the speculative thread executes. A less speculative thread (often the spawning thread) will stop execution when it arrives at an active CQIP of a more speculative thread and validates that control and data passed correctly from one thread to the other.

When a speculative thread is found to be no longer valid, the thread is squashed. The SpMT processor may squash a thread for several reasons. During the thread spawning event, register values and even memory inputs of the new speculative thread may be predicted by using a prediction algorithm [40, 25, 22, 32]. If values are incorrectly predicted, the error may be discovered when the less speculative thread tries to validate the predictions. In other cases, an invalid input value may cause an exception to occur, resulting in a squash. Additionally, unanticipated inter-thread memory dependencies are possible reasons to squash the thread. For example, if the parent thread writes to a memory block already read by the speculative thread, the speculative thread is executing incorrectly. When one thread is squashed its descendant threads are also squashed at whatever point they happen to be executing.

Branch prediction accuracy also affects the number of squashed threads. If a branch outcome is predicted incorrectly, the incorrect

path could encounter an SP. In this case, the processor spawns a speculative thread, but the thread is squashed when the correct branch target is resolved.

We make several assumptions about the underlying architecture, many of which are relatively orthogonal to the branch predictor problem. We assume little compiler support, to make the model more general, relying on structures easily recognized by hardware to identify spawning pairs – more specifically, we use loop continuations and procedure continuations. Thus, a procedure call becomes the SP, and the next static instruction (the return point) becomes the CQIP. On backwards branches, the branch target is both the SP and CQIP (allowing us to execute iterations in parallel). We assume support for fast thread spawn on another core without significant software overhead. If operating system code, for example, were executed to fork and set up a new thread, the GHR would naturally be filled (for better or worse) by the OS code. However, this type of overhead would prevent SpMT from being viable, and is not a reasonable assumption. Thus, the new thread begins executing user code at the CQIP without OS intervention.

Several other assumptions are made in some of our experiments to provide repeatable results for this study. In our simulations which model SpMT execution most faithfully, the results are highly noisy. For example, with several speculative threads running and competing for a limited number of idle cores, which thread was successful in spawning a new thread often depended on minor differences in performance. Thus, as we varied the branch performance, the actual threads that were spawned, and in what order and what frequency, varied widely from run to run.

Thus we make the following set of simplifying assumptions in most of the experiments in this paper, which provided very repeatable results, with the same threads being spawned, and each of those same threads having equal opportunity to impact performance in each experiment.

We assume an artificial 30% squash rate. In our SpMT framework, the best performance is achieved when liberal spawning policies produce a squash rate that is around 60%. However, to avoid overstating our results, and to account for other architectures that target more conservative spawning, we use the much lower value for these experiments. This conservative assumption, then, reduces the noise that is seen in the GHR. On the other 70% of spawns, we assume perfect register and memory prediction. A squashed thread is terminated at a relatively random location during its execution.

We also run the spawned threads serially. This again allows us to carefully run the same threads in the same order for repeatable experiments. It also provides a couple of very important analytical advantages – (1) it allows us to model an oracle GHR generator for comparison, because we can know the actual value of the GHR at the CQIP, and (2) it allows us to separate the performance of different classes of threads, since run serially the threads have an additive impact on performance. For example, we can look at the performance of short threads, medium threads, and long threads in separable groups. But we expect that the results are still highly indicative of the overall performance of a SpMT system, since we are running the same threads (that would execute and not be squashed).

But we also show the performance of our best prediction architecture in the context of a much more faithful model of speculative multithreading. Those results confirm the usefulness and performance of these techniques.

5. Methodology

This section describes our simulation framework, baseline processor core architecture, and benchmark selection.

Our underlying architecture implements a form of speculative multithreading. We modified a version of SMTSIM [46] to implement the described model. SMTSIM executes unmodified alpha ISA code and supports out-of-order SMT or CMP processors. In this study, SMTSIM is used to simulate execution of a CMP.

Cores	4	I cache miss penalty	17 cyc
Fetch width/core	4	D cache	64K, 2 way
INT instruction queue	64 entries	D cache miss penalty	17 cyc
FP instruction queue	64 entries	shared L2 cache	512K, 2 way
Reorder Buffer entries	256	L2 miss penalty	44 cyc
Branch Predictor	2Bc-gskew	L3 cache	4 MB, 2 way
Br mispredict penalty	21 cycles	L3 miss penalty	92 cyc
Cache line size	64 bytes	Fork penalty	10 cyc
I cache size	64K	Min spawn distance	20
I cache assoc	2 way	Max spawn distance	10000

Table 1. Architectural Specification

	BIM	G0	G1	META
prediction table	16K	64K	64K	64K
history length	4	13	21	15

Table 2. Characteristics of 2Bc-gskew predictor

Our framework simulates SpMT running on a parallel or multicore processor with the parameters listed in Table 1. We model 4 out-of-order cores, with shared L2 caches. Each core has a 2Bc-gskew branch predictor similar to the EV8 predictor [36], but without a few of the implementation idiosyncrasies, such as the shared hysteresis bits. The cores have a relatively deep pipeline, similar to a Pentium 4. More details of the branch predictor are given in Table 2.

2Bc-gskew [37] includes four different prediction tables, three of which are indexed with the GHR. Three of these tables are parts of an e-gskew [27] predictor (actually, two e-gskew tables and a bimodal table), and the fourth is the meta predictor. The meta predictor chooses between the results produced by the e-gskew predictor and the bimodal predictor. The three tables (two e-gskew and the meta table) use different indexing functions based on bitwise combinations of the GHR and PC.

The listed minimum and maximum spawn distance represents a filter that causes a thread not to be spawned if the expected distance between the spawn point and the CQIP is out of that range. Otherwise, a new thread will always be initiated at a spawn point if a core is available.

We choose eight benchmarks from the SPEC2000 suite. We intentionally select eight programs that are sensitive to the (overall) branch prediction accuracy in our simulation framework. We do this by filtering out those programs whose performance improved by less than 3% when a perfect branch predictor was introduced. We simulate 100 million instructions starting at a single execution Simpoint [38].

6. Generating Global History

To improve prediction accuracy and reduce destructive behavior caused by meaningless GHR values, several approaches will be considered. These various approaches fall into two broad categories. In the first category are techniques that attempt to predict or re-create the expected GHR, using current or historical information. The second category of techniques, conversely, only seek to provide a consistent starting point for the branch predictor every time the same thread starts up. These techniques will each be described in the following two sections.

All techniques will be compared with the *original* result, which retains whatever value was left in the GHR by the last thread which executed on that core.

Policy	New GHR Value
Original	No modification applied
Oracle	The GHR value at a given point as if the thread has been executed in a single core
Pre-spawn	The parent's GHR at the SP
Concat	The parent's GHR concatenated with the branch history from the SP to the CQIP of the parent thread when previously executed
Last-CQIP	The GHR at the CQIP of the last thread
Zero	0
PC	The PC of the CQIP
Xor	The PC $\oplus$ the parent's GHR

Table 3. GHR modification policies and their descriptions

6.1 Approximating the Real GHR Value

As an upper limit on the performance of this group we use the *oracle* policy. In SpMT, a speculative thread begins execution before all preceding branches are executed. The outcomes of these branches (the real GHR value) would be in the GHR at the start of the speculative thread in a perfect model. Although impossible to generate in real hardware, the simulator can be used to provide the real GHR to the speculative thread as if all the intervening branches had already been executed.

Although it is not possible to obtain the real GHR value, it should be possible to construct an estimate or a good proxy for the real GHR. The *pre-spawn* policy forces the speculative thread to inherit the GHR of the parent thread before it begins to execute. Since the parent's GHR at the SP does not contain any information regarding the future branches, this particular GHR value is incomplete. However, in many cases, this GHR will be unique to the SP, and thus indicative of the CQIP. In addition, by containing pre-SP path information, it may even usefully distinguish different instantiations of the thread. This can be particularly effective if the number of branches between the SP and CQIP are small. One downside to this policy is that it creates aliasing between instructions following the SP and those following the CQIP (which run immediately on different cores, but over time can execute on the same core).

The *concat* policy also utilizes the parent's GHR value at the SP, but it combines it with an estimate of the branch behavior between the SP and the CQIP. It does this by simply storing the bits from the last time this thread was spawned. It then concatenates the spawning thread's current GHR with the estimated history for the intervening branches. This policy improves the *pre-spawn* policy by referencing the missing branch outcomes. If there exists a dominant path from the SP to the CQIP of the parent thread each time these instructions are executed, the speculative thread has a good chance of emulating the real GHR exactly. If the program path behavior varies significantly from instance to instance (between the SP and CQIP), this policy no longer works in our favor.

The final policy of Group I is the *last CQIP* policy. This policy simply reuses the saved GHR from the last time a thread reached this CQIP. Thus, it uses old path data from both before and after the SP. This operation is a bit easier than *concat*; however, the dependence on the last execution of the spawn pair is greater, as it uses no path history from the current instantiation.

Although the *concat* and *last CQIP* are promising candidates to improve prediction accuracy, it could require significant storage to maintain the history from each SP/CQIP pair. In this study, we assume "large enough" structures to always provide the desired data without aliasing.

6.2 Providing a Consistent Starting Point

Our simplest mechanism is the *zero* policy. It simply clears the GHR when a new thread begins. The advantage of providing a consistent starting point is that it allows the predictor to start predicting reliably as soon as the GHR accrues enough history to distinguish it from other thread starting points. It also minimizes the amount of damage a polluted GHR can do to the prediction tables. However, the problem with the *zero* policy is that it creates aliasing among all threads at startup.

The *pc* policy provides a unique initial GHR state for each thread by generating the GHR bits from the PC of the speculative thread's CQIP. We first disregard the four least significant bits and use the remaining bits for the new GHR value (masked by the number of GHR bits desired). In an architecture with 32-bit fixed length instructions, the two least significant bits are meaningless. Furthermore, we ignore two more bits since the average basic block size is found to be greater than 4 [9]. The PC-based GHR provides a consistent starting point for all threads spawned from the same spawn point. This policy will be beneficial particularly if the branch pattern at the beginning of the thread is consistent (and not heavily dependent on the path before the CQIP). If this is true, this technique can even outperform the real GHR.

Finally, the *xor* combines a predictor from each group. This policy adds some path information to the previous policy, which can help if the branch behavior of the new thread is more heavily path dependent. In this case, we use the parent's GHR, as used in the *pre-spawn* policy above. The *xor* policy combines the PC and the parent's GHR by using the XOR operation.

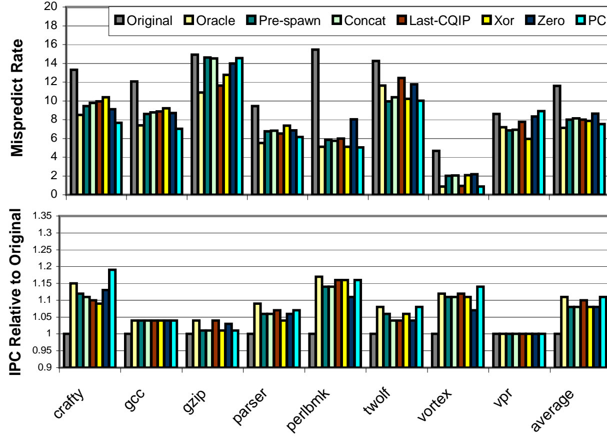
7. Results

In this section, we evaluate the branch prediction accuracy with our generated GHR values. We demonstrate that most policies improve the prediction accuracy and also result in IPC improvement.

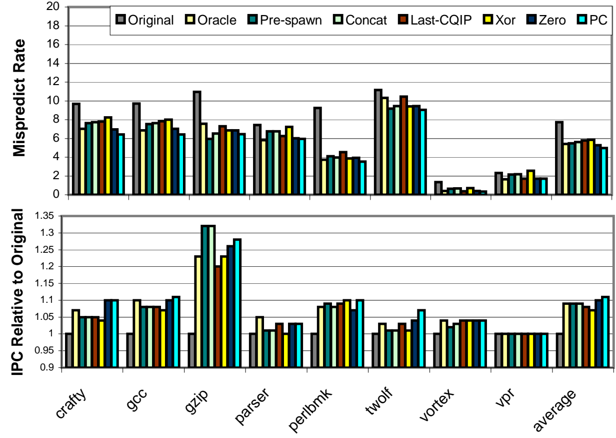
7.1 GHR Correlation

Ultimately, our goal is to replace a GHR that contains noisy data (which is not expected to be well correlated with the behavior of the new thread) with data that correlates more closely to the new thread's branch behavior. We study that phenomenon in this section – the correlation of potential GHR values with the branch behavior of the spawned thread.

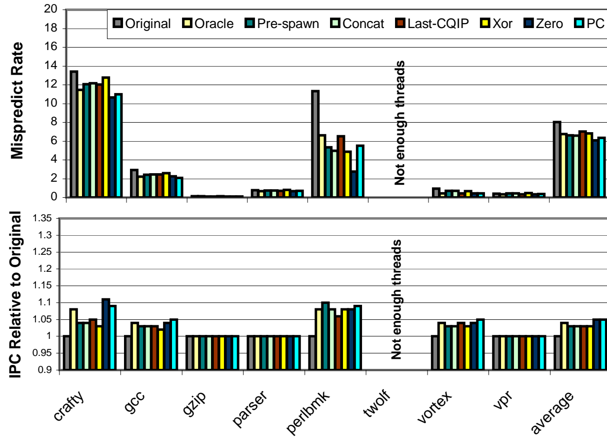
We did extensive studies of the correlation of the GHR (real or fabricated, according to our various proposals) to the branch behavior at the beginning of a new thread. We will just summarize those results here. For simplicity, we will use a crude definition of correlation. For the average GHR in place at the start of a new thread, what is the frequency of the most likely branch path observed when the thread began with that GHR (the branch path must be identical out to 21 branches)? For example, assume we saw three different paths, A with 50% frequency, and B and C each with 25% frequency. Assume we also observed two initial GHRs, X (which always resulted in path A) and Y (which resulted in B half the time and C half the time). Since X and Y occur with equal



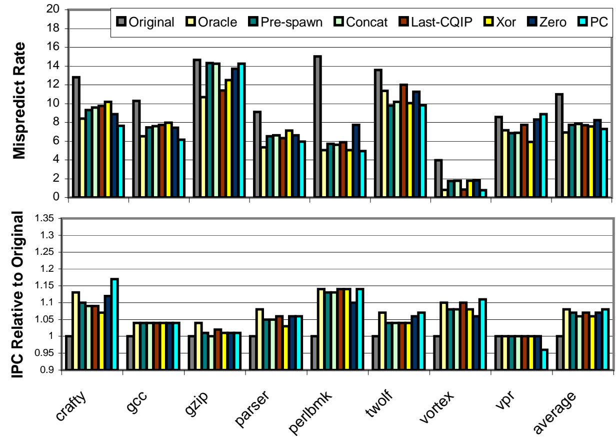
(a) Threads less than 500 instructions



(b) Threads with between 500 and 2000 instructions



(c) Remainder (more than 2000 instructions)



(d) Overall

Figure 3. Branch prediction accuracy and relative IPC to Original for threads with committed instructions (a) less than 500, (b) less than 2000 but greater than 500, (c) greater than 2000, and (d) average over all lengths.

frequency, the average correlation would be 75%, composed of X's 100% and Y's 50% frequency in seeing their most likely branch path. For the *zero* predictor, which only supplies a single GHR to all threads, it would record a 50% correlation for the same set of paths.

For the *original* case, the correlation is surprisingly high. We observe nearly a 50% correlation between the starting GHRs and the most likely paths. But most of our predictors do better. The oracle achieves nearly 70%. Even more surprisingly, the *pc* predictor matches the correlation of the oracle. This is despite the fact that the oracle can potentially differentiate both different threads and a different spawn context (in terms of branch history) for those different threads. The *pc* predictor can only differentiate different threads, and has no knowledge of the context in which a thread was spawned. The mechanisms which manufacture GHRs (*prespawn, concat, last-ghr*) all have lower correlations. The *zero* predictor, which cannot even differentiate between threads, has the lowest correlation, less than 30%.

These results, particularly the correlation data of the *pc* predictor, imply that at the CQIP points we are selecting, the pre-CQIP branch history is not overly important in predicting post-CQIP branches.

7.2 Branch Prediction Accuracy

Figure 3 shows branch prediction accuracy and IPC speedup for all of our different policies. These results are shown in four parts. In Figure 3(a), accuracy and performance for very short threads (<500 instructions) is shown. The *oracle* and *pc* policies perform the best for short threads and, in fact, our *pc* model performs as well as *oracle*. These reduce mispredictions by 35% and 38% respectively and provide an 11% IPC speedup on those threads alone.

It is not altogether surprising that resetting the GHR to a thread-specific value can even beat the oracle predictor (which uses the full, correct, GHR) in a few of our benchmarks. If the path after the CQIP is not highly correlated with the path before the CQIP, setting the GHR to a single value at the CQIP is actually better. In our thread-spawning model, we tend to place the CQIP in locations where the correlation *may* be minimized – at procedure boundaries and loop iteration boundaries. This indicates that expensive efforts to try to predict the exact GHR at these boundaries is unnecessary.

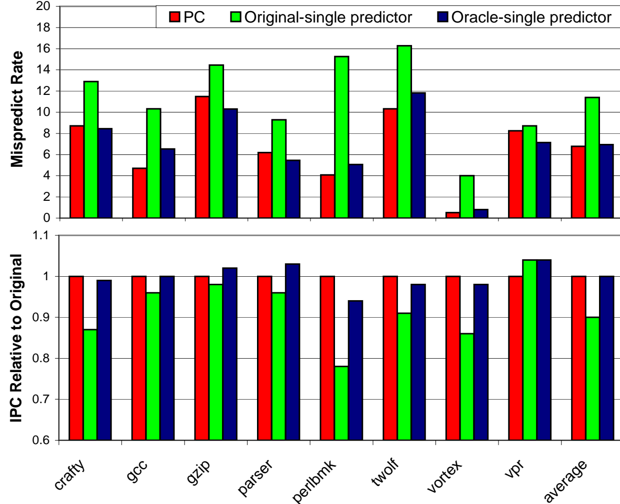


Figure 4. Comparison of the *pc* policy against idealized branch predictor as if all cores share one predictor.

For SpMT systems with a more general thread spawning strategy [32], this result may still hold. This is because a good CQIP will tend to be control flow independent of prior branches, and also represent an execution discontinuity in that it seeks to minimize the number of values live across the CQIP [23].

Figure 3(b) shows the same results, but for threads less than 2000 instructions (but longer than 500). In these results, the same effects appear but are dampened by the fact that the GHR now has time to fully warm up. In this range, both the *pc* and *zero* policies are essentially equivalent to the oracle. Similar to threads less than 500 instructions, the *pc* policy reduces mispredictions by 36% and achieves a 10% IPC speedup.

Even for the large threads (Figure 3(c)) there are non-trivial differences between our techniques, and a clear advantage to having a policy to synthesize the GHR. The gains, of course, are smaller. *Zero* and *pc* provide a 7% and 6% speedup respectively.

Figure 3(d) provides the actual overall results. The *oracle* policy, because of its strong performance with short threads, performs better than the *pc* policy, which clearly outperforms all of the realistic approaches. The *oracle* policy provides a 37% reduction and the *pc* policy provides a 34% reduction in mispredicted branches. Each policy provide a 9% IPC speedup. *Crafty* obtains the strongest improvement from *pc* – a 17% speedup. The *pc* policy improved *Perl* by 15% although most of the proposed policies did equally well. *Vpr* is the only benchmark for which the *pc* policy hurts performance.

So despite our efforts to construct a reasonable GHR, we find that the simple *pc* policy consistently beats those techniques. The intuition is as follows. For threads whose control flow behavior is relatively independent of the pre-CQIP branch history, using a single value is at least as good as using the real history. On the other hands, for those threads whose control flow is highly correlated to the pre-CQIP history, a manufactured history that uses obsolete data is likely to do more harm than good.

Another useful insight from these results comes from the fact that the *pc* predictor is competitive with, and in some specific cases (especially when considering individual threads) better than the *oracle* predictor. This implies that in some cases the GHR contains much more information than is needed. When branches are no longer correlated with the history, the diverse values in the GHR only make it harder to train, and adds pollution to the predictor.

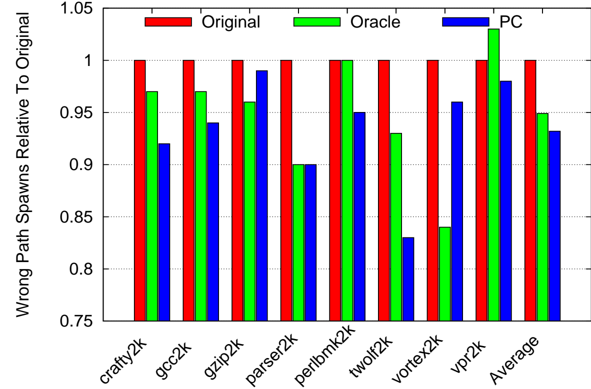


Figure 5. Threads spawned while the processor is executing incorrectly predicted paths

We can possibly apply this even in situations where threads are not being migrated or spawned. If we can identify those points (where there is some kind of break in correlation) and set the GHR to a PC-specific value, we should be able to improve the accuracy of any of the GHR-based predictors. Likely points where this policy might be employed would be procedure boundaries and loop exits – because there is more likely to be a discontinuity (low correlation) between the preceding path and the following path. This is the subject of future research.

The fact that the *pc* policy is the clear winner is fortuitous for several reasons. First, the cost of implementation is effectively none. Even *pre-spawn* would have a small cost, as the spawning thread would need to send it's GHR to the new core. But the pc policy has effectively no cost because the pc must already be sent to start the thread. Second, the PC scheme will translate naturally to our other target environments – helper threads and short compiler-generated threads – and should perform equally well there. It is not clear that the schemes that use, for example, pre-spawn path would be as useful in those environments, particularly for the compiler-generated threads.

Thus, we have addressed all of the discussed sources of short threads. For long threads with frequent migration, we have shown that simply migrating the GHR is sufficient. For speculative multithreading, we set the GHR to a thread-specific value (derived from the program counter). We have not specifically demonstrated short helper threads or short compiler-generated threads, but because those environments will share the most important features of our environment – short threads, no clear "correct" value for the GHR, and spawned execution often independent of the code that preceded the point of spawning – we expect the *pc* technique to do well.

7.3 Transferring Branch Predictor State

We showed evidence in Section 2 that the problem with distributed branch prediction is not the prediction table state, but rather with the global history state contained in the GHR. We explore that thesis more carefully here. We model a scheme that transfers over the entire predictor state (for each of the tables in the 2Bc-gskew predictor) when a thread is spawned – also in an oracle manner, using the actual state at the CQIP. However, this scheme does not solve the GHR problem. Additionally, we do not charge for the cost of transferring all of this data. We call this the *original-single predictor* model. We compare it with our *pc* model (now called *pc-unshared*) for clarity, and another policy which transfers all predictor state and the oracle GHR – the *oracle-single predictor*. We use

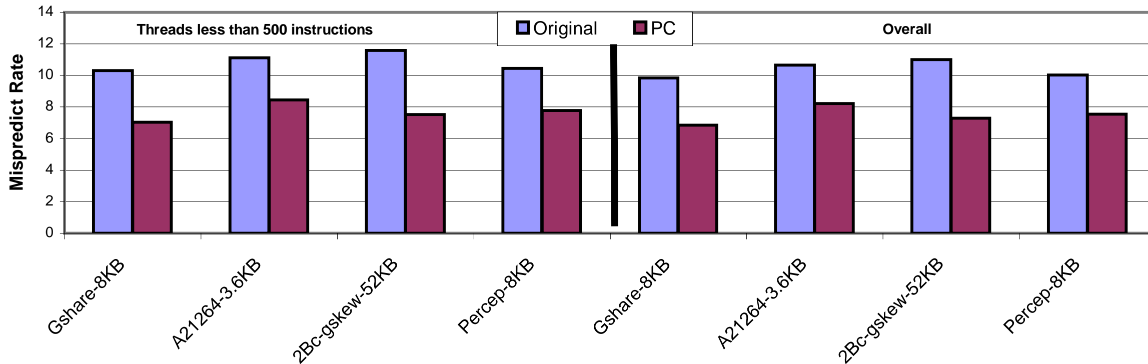


Figure 6. Mispredict rate for examined branch predictors using the baseline and our *pc* policy. Results for short threads (less than 500 instructions) are on the left and overall results on the right.

the term "single predictor" because transferring all predictor state is very similar to using a single predictor.

In Figure 4 we see that the cost of not addressing the GHR issue is very high, and the ability to transfer predictor state does not overcome that handicap. But even further, we see that (comparing the pc and oracle results), transferring the full predictor state actually has *no value*. This is a result of the tradeoff between losing some history information and being able to gain some predictor space (across the four cores).

7.4 Wrong Path Spawns

By simplifying our simulated SpMT framework and trying to run deterministic, repeatable experiments, we simplified away one of the key advantages of accurate branch prediction in our SpMT architecture. In this architecture, we spawn threads when we fetch an SP instruction, but for the purposes of this study we did not cause a spawn to happen when the SP was encountered on the wrong path. In a real architecture, however, that would happen (unless spawn was delayed until commit, which would significantly increase the observed thread startup overhead). In this section, we seek to understand the impact of our branch prediction mechanisms on wrong path spawns.

The primary cost of a wrong path spawn is the opportunity cost of not having that core (and possibly other cores running threads spawned by the incorrectly spawned thread, and so on) available to execute correct-path threads. An additional overhead is the cache pollution in the new core resulting from the execution of the incorrectly spawned thread. Even if the mispredict is discovered quickly, the communication cost to announce the squash and the time to drain the pipeline will still result in a loss.

Wrong-path prediction of speculative threads has been presented in [42], but we do not assume that hardware support here. We present preliminary results given our SpMT model in Figure 5 which demonstrates that branch prediction using our *pc* policy substantially reduces wrong-path spawns.

7.5 Sensitivity to the Branch Predictor

All results up to this point have been shown using a single branch predictor, in particular a branch predictor with heavy reliance on the GHR. In this section, we examine the effectiveness of our best technique in the presence of other branch predictors. In addition to the 2Bc-gskew predictor, we also model the gshare predictor [26], the Alpha 21264 tournament predictor [18], and the perceptron predictor [17]. Figure 6 demonstrates that while indeed the EV8-like predictor is most sensitive to the short-thread GHR problem, all predictors are sensitive to a significant degree, and all see their prediction accuracy improved.

The perceptron predictor, as previously noted, is able to predict in the presence of noise. However, because it uses a longer GHR, it still takes some number of branches before the noise is overcome by the meaningful bits in the GHR. The perceptron predictor modeled here uses a 28-bit GHR.

It must be noted that all of these predictors are still heavily influenced by the path discontinuities inherent to a speculative multithreading architecture. Although the influence is greatly reduced by our solutions, it still remains even in those results. One evidence of this is the fact that the magnitude of the predict rates (both with and without our solutions), and even the relative ordering of the predictors is different than what the literature would lead one to expect. This is because the performance of these predictors on a speculative multithreading architecture is heavily influenced by how well each predictor adapts to these discontinuities, and that adaptability varies by predictor.

7.6 Full Speculative Multithreading

We greatly simplified our speculative multithreading simulation infrastructure to induce repeatability in the experiments and to facilitate analysis of the data. Those results indicate significant gains in predict rate and execution rate for both short and medium-length threads. This section confirms that those results hold for a more faithful speculative multithreading implementation.

In this section, we allow threads to run in parallel to the extent that the code allows, and we squash threads on actual memory violations (rather than probabilistically). We use the same spawning points and control-quasi independent points. We do, however, assume a somewhat more aggressive system that allows a thread to be spawned if the expected overlap between the main thread (before reaching the CQIP) and the speculative thread is greater than 10 instructions. This would be appropriate for a system where the thread initiation overhead is low – a reasonable assumption since that is the goal of this research. The resulting average thread length for these simulations is about 200 instructions. We no longer assume memory dependence prediction, but squash on any memory violation between threads. Thus, the GHR is a product of the last committed instructions from the previous completed or squashed thread on that core. We are again using the EV8 predictor.

The results are shown in Figure 7, which compares a system that makes no special accomodation for the GHR (representing virtually all former SpMT results, in that respect) and one that uses the PC to initialize the GHR when a new thread is spawned. We see that these results do track fairly well with the prior results. *Crafty* gets

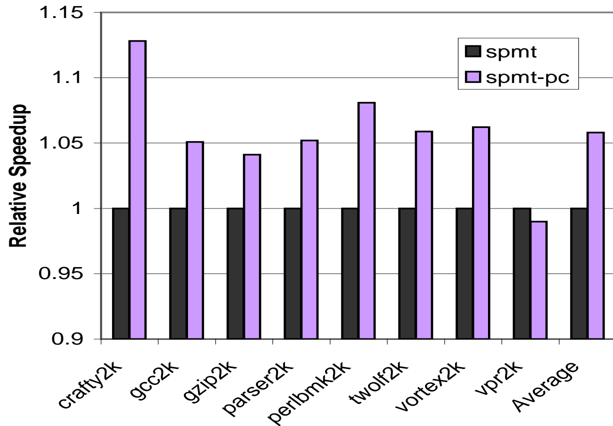


Figure 7. Performance of a full speculative multithreading framework with and without our best predictor.

the most benefit from the *pc* technique, with a 13% improvement. *Vpr* is still the only benchmark that does not respond well. Overall, we see a 29% overall reduction in branch mispredicts with our best predictor.

Given the cost of our GHR synthesis solution (none), and the tangible gains for a wide variety of advanced predictors, this solution is an obvious choice for any architecture that has the ability to generate threads quickly and often, without OS overhead, on another core.

Again, this models a specific SpMT implementation, with accurate register prediction, no memory prediction (but confidence counters to avoid spawning frequently squashed threads), and no compiler support for improved placement of SPs and CQIPs. However, we expect these results to be relatively general, as the problem of frequently starting short threads without accurate GHR data is fairly common in the various proposed architectures. Even with compiler support for identifying CQIPs, we would tend to place them at points of discontinuity, where the *pc* policy is likely to be most effective.

8. Conclusion

This paper demonstrates that significant branch predictor state is lost when control transfers across cores or contexts in a multiprocessor or multithreaded architecture. If those threads do not execute long, the cost of the resulting branch mispredicts is never amortized. In particular, we model a speculative multithreading architecture running on a CMP, aggressively spawning threads.

We show that the critical loss of state lies entirely with the global branch history, as stored in the GHR. The branch predictor state stored in the predictor tables themselves is of no account, because if the program runs long enough, each core's tables will acquire that state. This implies that we can transfer that state extremely quickly. However, we further show that the best solution does not transfer any global history, but rather synthesizes a value in the new core's GHR in such a way that it allows each unique thread to update and use it's own branch history immediately. We do this by constructing the initial GHR out of the meaningful bits of the program counter at the thread's starting address.

This results in significant (relative to the cost) performance gains on a realistic speculative multithreading framework. We show this both for a somewhat constrained SpMT simulation environment that gives us repeatable results and allows us to analyze the data carefully, and also for a more realistic SpMT environment with threads running in parallel and being squashed based on actual execution characteristics.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported in part by NSF Grant CCF-0541434 and Semiconductor Research Corporation Grant 2005-HJ-1313.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *31st International Symposium on Microarchitecture*, pages 226–236, Nov. 1998.

[2] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *32nd Annual International Symposium on Computer Architecture*, pages 298–309, June 2005.

[3] P. Chaparro, J. Gonzalez, and A. Gonzalez. Thermal-aware clustered microarchitectures. In *International Conference on Computer Design*, pages 48–53, Oct. 2004.

[4] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *26th Annual International Symposium on Computer Architecture*, pages 186–195, May 1999.

[5] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, Oct. 1996.

[6] J. Chung, H. Chafi, C. Minh, A. McDonald, B. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *Sixth International Symposium on High-Performance Computer Architecture*, pages 266–277, Feb. 2006.

[7] M. Cintra, J. F. Martńez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

[8] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[9] M. de Alba and D. Kaeli. Path-based hardware loop prediction. In *4th International Conference on Control, Virtual Instrumentation and Digital Systems*, August 2002.

[10] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *2th Annual International Symposium on Computer Architecture*, pages 126–132, June 1975.

[11] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *31st International Symposium on Microarchitecture*, pages 69–77, Nov. 1998.

[12] J. Gummaraju and M. Franklin. Branch prediction in multi-threaded processors. In *9th International Conference on Parallel Architectures and Compilation Techniques*, pages 179–188, Oct. 2000.

[13] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, Oct. 2004.

[14] S. Hily and A. Seznec. Branch prediction and simultaneous multithreading. In *Conference on Parallel Architectures and Compilation Techniques*, page 169, Oct. 1996.

[15] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–348, Feb 1996.

[16] D. A. Jiménez. Fast path-based neural branch prediction. In *36th International Symposium on Microarchitecture*, page 243, Dec. 2003.

[17] D. A. Jiménez and C. Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, Feb. 2002.

[18] R. E. Kessler. The alpha 21264 microprocessor. *IEEE MICRO*, 19(2):24–36, Mar. 1999.

[19] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In *36th International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[20] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *37th International Symposium on Microarchitecture*, pages 195–206, Dec. 2004.

[21] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-driven multithreading using conventional microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1176–1188, Oct. 2006.

[22] P. Marcuello. Speculative multithreaded processors, Ph. D. Thesis, Universitat Politecnica de Catalunya. 2003.

[23] P. Marcuello and A. Gonzélez. Thread-spawning schemes for speculative multithreading. In *Second International Symposium on High-Performance Computer Architecture*, page 55, Feb 2002.

[24] P. Marcuello, J. Tubella, and A. Gonzélez. Value prediction for speculative multithreaded architectures. In *32nd International Symposium on Microarchitecture*, pages 230–236, Nov. 1999.

[25] P. Marcuelo and A. Gonzàlez. A quantitative assessment of thread-level speculation techniques. In *14th International Symposium on Parallel and Distributed Processing*, page 595, May 2000.

[26] S. McFarling. Combining branch predictors. *DEC WRL Technical Note TN-36*, 1993.

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

[28] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the Hydra CMP. In *13th International Conference on Supercomputing*, pages 21–30, June 1999.

[29] G. D. Pizzol and P. O. A. Navaux. Branch prediction topologies for SMT architectures. In *Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pages 118–125, June 2005.

[30] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *10th Symposium on Principles and Practice of Parallel Programming*, pages 142–152, June 2005.

[31] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *33rd International Symposium on Microarchitecture*, pages 269–280, Dec 2000.

[32] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Conference on Programming Language Design and Implementation*, pages 269–279, June 2005.

[33] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *16th Annual International Symposium on Computer Architecture*, pages 46–53, Apr 1989.

[34] A. Seznec. Analysis of the O-GEometric history length branch predictor. In *32nd Annual International Symposium on Computer Architecture*, pages 394–405, 2005.

[35] A. Seznec. The L-TAGE branch predictor. In *Journal of Instruction-Level Parallelism, vol. 9*, May 2007.

[36] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha EV8 conditional branch predictor. In *29th Annual International Symposium on Computer Architecture*, pages 295–306, June 2002.

[37] A. Seznec and P. Michaud. De-aliashed hybrid branch predictors. *Technical Report RR-3618, Inria*, Feb. 1999.

[38] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

[39] J. E. Smith. A study of branch prediction strategies. In *25th Annual International Symposium on Computer Architecture*, pages 202–215, June 1998.

[40] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[41] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor: a mechanism for reducing negative branch history interference. In *24th Annual International Symposium on Computer Architecture*, pages 284–291, June 1997.

[42] S. T. Srinivasan, H. Akkary, T. Holman, and K. Lai. A minimal dual-core speculative multi-threading architecture. In *International Conference on Computer Design*, pages 360–367, Oct. 2004.

[43] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *4th International Symposium on High-Performance Computer Architecture*, page 2, Jan. 1998.

[44] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[45] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, Sep. 1999.

[46] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *In 22nd Annual Computer Measurement Group Conference*, December 1996.

[47] T. Vijaykumar, S. Gopal, J. Smith, and G. Sohi. Speculative versioning cache. *IEEE Transactions on Parallel and Distributed Systems*, 12(12):1305–1317, Dec. 2001.

[48] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *12th Symposium on Principles and Practice of Parallel Programming*, pages 79–89, Sep 2007.

[49] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.

[50] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *14th International Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, Sep. 2005.

[51] W. Zhang, B. Calder, and D. M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *International Symposium on Code Generation and Optimization*, pages 50–64, March 2006.

[52] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. *SIGARCH Computer Architecture News*, 29(2):2–13, June 2001.