1-1-2010

Broadcast Mechanism for improving Conditional Branch Prediction in Speculative Multithreaded Processors

Renjith Thankappan Achary Retnamma
Portland State University

Recommended Citation

Thankappan Achary Retnamma, Renjith, "Broadcast Mechanism for improving Conditional Branch Prediction in Speculative Multithreaded Processors" (2010). *Dissertations and Theses.* Paper 368.
https://doi.org/10.15760/etd.368

Broadcast Mechanism for improving Conditional Branch Prediction in Speculative Multithreaded Processors

by

Renjith Thankappan Achary Retnamma

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Thesis Committee:
Douglas V. Hall, Chair
Dan Hammerstrom
Christof Teuscher

Portland State University

ABSTRACT

Many aspects of speculative multithreading have been under constant and crucial research in the recent times with the increased importance in exploiting parallelism in single thread applications. One of the important architectural optimizations that is very pertinent in this scenario is branch prediction. Branch Prediction assumes increased importance for multi-threading systems that execute threads speculatively, since wrong predictions can be much costlier here, in terms of threads, than a few instructions that occupy the pipeline in a uni-processor. Conventional branch prediction techniques have provided increasingly better prediction accuracies for uni-core processing. But the branch prediction itself takes on a whole new dimension when applied to multi-core architectures based on Speculative Multithreading. Dependence on global branch history has helped branch predictors to achieve high prediction accuracy in single thread applications. The discontinuity of global history created at the thread boundaries cripple the performance of branch predictors in a multi-threaded environment.

Many studies in the past have tried to address the branch history problem to improve the prediction accuracy. Most of these have been found either to be architecture specific or complex in terms of the hardware needed to recreate or approximate the right history to be given to the threads when they start executing out of order. This hardware overhead increases as the number and size of threads increase thereby limiting the scalability of the algorithms proposed so far. The current thesis takes a different direction and proposes a simple and scalable solution to effectively reduce the misprediction rates in Speculative Multithreaded systems. This is accomplished by making use of a synergistic interaction between threads to boost the inherent biased nature of branches and using less complex hardware to reduce aliasing between branches in the threads. The study proposes a new scheme called the Global Broadcast Buffer scheme to effectively reduce branch mispredictions in Speculative Multithreaded architectures.

Dedicated to my parents Thankappan and Retnamma

Acknowledgements

I thank my guide Prof. Haitham Akkary for giving me the right direction and focus throughout the thesis work. I am grateful to my advisor Dr. Douglas V Hall, who at every stage of the thesis, made sure that I was on track by checking the status and giving the right advice. He gave me invaluable insight and tips in achieving perfection in the work done. I thank the thesis committee members Dr. Dan Hammerstrom and Dr. Christof Teuscher for providing valuable feedback and suggestions on my work. Without the presence of support of my friends Komal Jothi, Manjith, Vishwa, Vinay, Srini, Naveen Balasingam and Arun Veera I would not have been able to accomplish this task. Viswesh, Ptd, Rosh and Madhu, who have been a constant source of encouragement, kept their faith in me time and again throughout this research work. They deserve more than a thank you. Finally I thank The Almighty, who I believe, was there with me always and gave me the patience, perseverance and courage in finishing my work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Motivation

1.1 Introduction

Various methods to improve the computation speed of processors have been under development in the recent years. Amdahl's law [28] provided architects with the basic constraints in speeding up the pipeline and also the further thought processes that would lead to many innovations in the field of processor architecture, witnessed in the years that followed, and still continuing. As the complexity of applications increased over time, the need to speed up the execution demanded more out of the pipelines that were sticking to in-order execution. The key was to find and extract parallelism from the sequential applications. Evolution of various software and hardware methodologies to speed up execution by using the parallelism in programs gave rise a whole new area of research in the field of computer architecture, called Parallel Computing.

Parallel Computing uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above. The endeavor to extract available parallelism started at the instruction level where many instructions in a single program were found to be independent of each other and could be executed in parallel. This transformed the traditional single fetch-execute model into the superscalar pipeline which achieved parallelism using multiple issue using dynamic scheduling. A series of new methods were devised at the compiler level [25] to support this and provide static issue pipelines with parallelized stream of instructions within a single program.

The first and the basic trials to parallelize execution were to find and exploit the available parallelism among instructions (ILP). Even in a purely sequential program there are many places where the contiguous instructions are data and control independent. Finding out this dependence, resolving it and allowing instructions to execute out of order brought in many new stages to the conventional five stage pipeline [25]. All these techniques, while increasing the pipeline depths to double or triple the conventional depths, increased the importance and criticality of branch prediction. The reason was simple. On a misprediction, the speculatively fetched or executed but not committed instruction needed to be flushed out of the pipeline and fetching needed to start from the right path. This penalty was significant as the pipeline complexity and depths shot up with innovations in architecture as well as silicon technology. As architectures evolved from uni-processor to multicores, where parallelism was found and exploited not only in terms of instructions but a chunk of instructions (called threads), the importance of branch prediction rose to a level which needed a different approach in the design of branch predictors. In the context of various threads running at the same time speculatively, a misprediction could cause a thread flush, which is quite expensive in terms of both power and execution time. In the following sections the basic branch prediction techniques are discussed followed by an introduction to Speculative Multithreading (SpMT).

1.2 Branch Prediction

In simple terms, branch prediction is a mechanism which helps in speeding up execution in the processor pipeline without waiting for the branch instruction to execute and the outcome is known. Branch prediction is very critical in utilizing the instruction level parallelism (ILP) available in the applications in modern pipelines. Branches can be divided into two basic types, conditional and unconditional. Conditional branches take the course of execution in a direction which depends on the evaluation of a certain condition. An "if-else" statement is a very common example of a conditional branch. An unconditional branch takes the course of execution to a point without depending on the evaluation of any condition. Both conditional and unconditional branches are encountered very frequently in applications. In this thesis the focus is on conditional branches and how well they are predicted in a multicore environment.

Branch prediction mechanism consists of two parts

1) Predicting the direction that a branch will take

2) Fetching of the instructions into the pipeline from the predicted path. The branch prediction mechanism provides the decision whether a branch is taken or not taken (direction) and a buffer called the Branch Target Buffer (BTB) provides the instructions in the predicted path.

First of all, let us examine why branch prediction is an important aspect in pipeline design and how it helps improve the pipeline throughput. Consider the classic five stage pipeline (MIPS) [25] where an instruction goes through five stages, fetch, decode, execute, memory and write back. While an instruction is being executed, a later instruction in the program is being decoded and another one being fetched. Now consider a branch instruction being fetched. A branch instruction takes two paths, taken or not taken. We say a branch is not taken if the condition evaluated turns out to be false and the execution falls through to the sequential instruction in the program flow after the branch. If the condition is evaluated to be true, then the fetching should start from the instruction pointed by the new value of the instruction pointer, which is the taken address or the target. But whether to fetch the non-taken instruction or the target will be known only after the branch instruction is decoded and the outcome of the instruction is known. This leads to something we term as control hazard in pipeline. In other words this leads to useless pipeline cycles where we wait for the branch instruction to give the correct address of the next instruction to be fetched. In the classic five stage pipeline, one pipeline cycle is wasted on every conditional branch. The instruction after the branch in always fetched and then ignored. The fetching resumes again with the resolved address when the branch instruction is decoded. In the case of a non taken branch, the second fetch of the following address is redundant. In the absence of any optimization, one clock cycle is wasted in the pipeline. One stall cycle for every branch will result in a performance loss of 10% to 30% depending on the branch frequency.

1.2.1 Static and Dynamic Branch Prediction

Very early solutions to overcome branch hazards in pipeline were to assume that the branches are either taken or not taken all the time or to use delayed branches. We call this static branch prediction. Delayed branches inserted a useful instruction into the pipeline right after the branch instruction so that irrespective of the branch outcome the delayed instruction will get executed. These methods were of use when dealing with smaller pipelines where the resolution of the branch target didn't need a lot of pipeline cycles. But as the processors moved from simple in-order to superscalar out-of-order [1] with deeper pipelines, the penalty of going the wrong way on a misprediction became significant. Delayed branches couldn't be used in such pipelines due to the difficulty in finding enough instructions to fill in the delay slots. The static methods did not take into account the runtime behavior of branches that became crucial in achieving high accuracy in more complex, deeper and wider pipelines. Dynamic branch predictors make use of the runtime behavior of branches to make predictions in a pipeline.

1.2.2 A summary of Dynamic branch prediction schemes

Dynamic branch prediction is a hardware based prediction scheme. The most well known technique, referred to as bimodal branch prediction, makes a prediction based on the direction the branch took, the last few times it was executed. More recent work has shown that significantly more accurate predictions can be made by utilizing branch history. One method considers the history of each branch independently and takes advantage of repetitive patterns. Since the histories are local to each branch, we will refer to it as local branch prediction. Another technique uses the combined history of the recent branches in a program to predict a specific branch. This technique will be referred to as global branch prediction. Each of these different branch prediction strategies has distinct advantages. The bimodal technique works well when each branch is strongly biased in a particular direction. The local technique works well for branches with simple repetitive patterns. The global technique works particularly well when the direction taken by sequentially executed branches is highly correlated.

A bimodal predictor is implemented using a table of saturated counters indexed by a specific number of low order bits of the branch address. The value of the saturated counter predicts which direction the branch should take. The behavior of typical branches is far from random. Most branches are either usually taken or usually not taken [25]. Figure 1.2.2.1 depicts a bimodal predictor. T and NT denote taken and not taken directions.

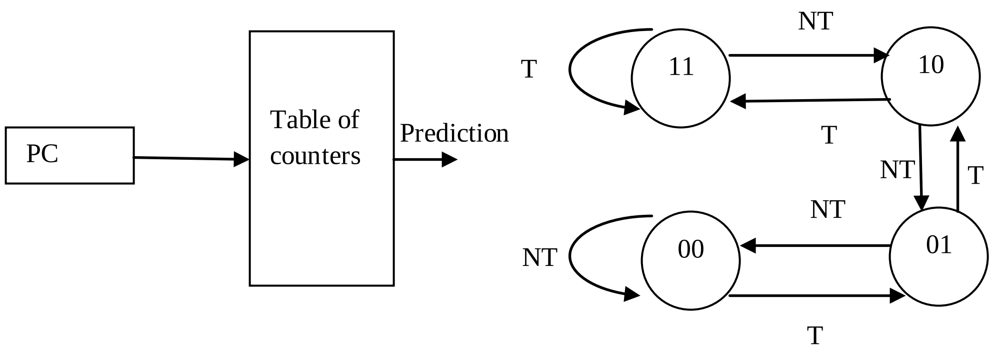

Figure 1.2.2.1 Bimodal Branch Prediction

A typical local branch prediction scheme is shown in figure 1.2.2.2. There are two levels of tables. A history table and a pattern history table (PHT). Each history table entry records the direction taken by the most recent N branches whose addresses map to this entry, where N is the length of the entry in bits. The PHT is an array of 2-bit counters identical to those used for bimodal branch prediction and are indexed by the branch history stored in the first table.

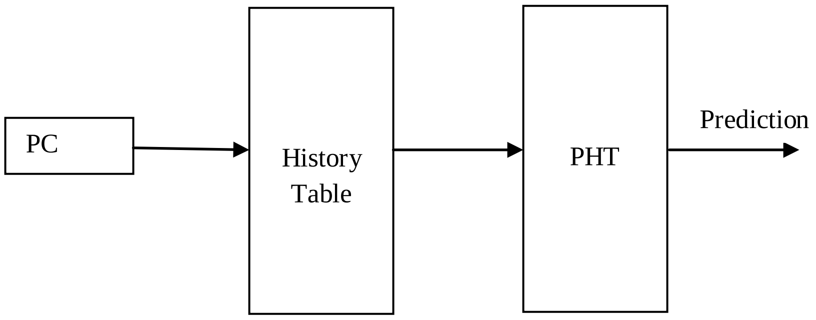

Figure 1.2.2.2 Local Branch Prediction

Global Branch Prediction comes in two flavors

1) Global History with Index Selection (gselect).

2) Global History with Index Sharing (gshare).

In the index selection scheme (figure 1.2.2.3.a) a specific number of least significant bits of the branch address is concatenated with the information in the history register to form an index to the saturating counters. This scheme provided more specificity to a particular branch in relation with the global history to provide better prediction. The main drawback of this scheme was the scalability. For increasing the unique correlation between a branch and the global history, more bits needed to be used and this massively increased the size of the PHT. The solution was the index sharing scheme (gshare) (figure 1.2.2.3.b) [2]. In this scheme the history bits are XORed with an equal number of least significant bits of the branch address to create an index to the prediction table.

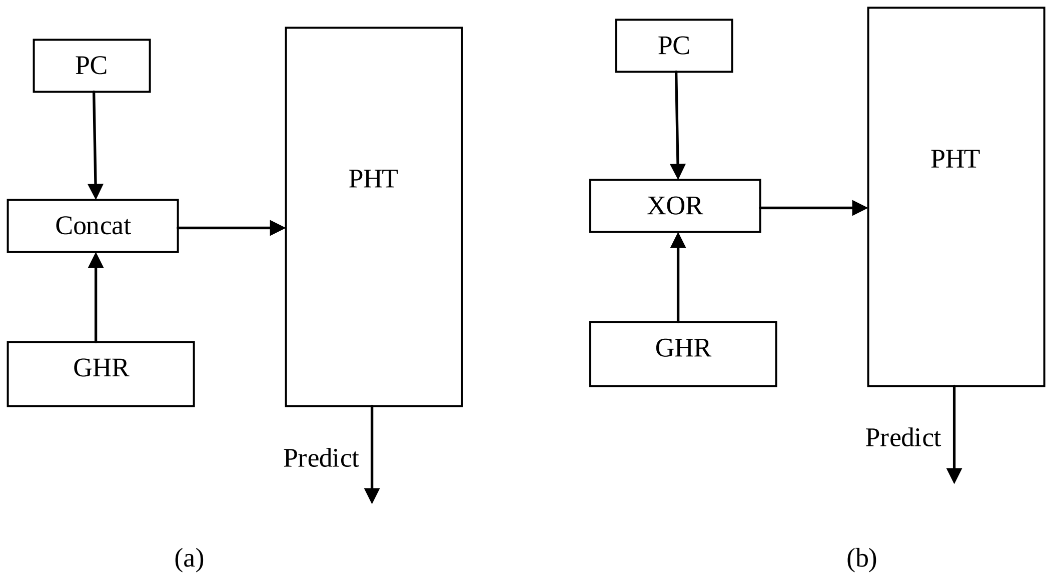

Figure 1.2.2.3 gselect and gshare configurations of global branch prediction

1.3 Speculative Multithreading

Speculative Multithreading (SpMT) divides a single thread application into multiple threads which are executed speculatively out of order on different cores to improve performance. A typical SpMT system will have one non-speculative thread and one or more speculative threads at any point of time [18]. These threads run 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 thread level parallelism in a program.

Threads are speculative since they are data and control dependent on previous threads. If speculation is correctly performed, the execution time of the application is drastically reduced due to the additional exploitation of speculative thread-level parallelism. However, if a misspeculation occurs [18], roll-back mechanisms are necessary to return the processor to a correct state. Multi-Processors that are able to execute speculative threads are referred to as Speculative Multithreaded Processors. A speculative multithreaded processor consists logically of replicated processing elements that cooperatively perform the parallel execution of threads. Speculation is key: Without speculation, we can only divide programs conservatively into nonspeculative threads whose mutual independence and execution is guaranteed. Speculation enables more aggressive divisions that can exploit threads whose independence and execution may not be guaranteed but are parallel, and likely to be executed, with high probability.

In a typical SpMT model, a non-speculative thread runs first and spawns a speculative thread when it hits the Spawning Point (SP) [29]. The SP is an instruction that triggers the processor to create a new speculative thread. The speculative thread starts running from an instruction which is control independent of the SP. This instruction is called the Reconvergence Point (RIP). The RIP is the first instruction that the speculative thread executes. Each SP is mapped to a RIP. The non-speculative thread will stop execution when it arrives at its RIP and validates that control and data is passed correctly from one thread to the other. The oldest speculative thread then becomes the non-speculative thread. When a speculative thread is found to be no longer valid, the thread is squashed. Branch prediction accuracy 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.

1.4 Motivation

1.4.1 What are we trying to solve – The history problem

Branch Prediction assumes greater importance in the multithreaded-multicore scenario. This is because of the fact that the concept of how we predict branches changes to a big extent from the traditional sequential flow of execution in a uniprocessor. The methods to predict branches in a uniprocessor have been developed over time by making use of the data available at any point of execution, from the past, in other words, the history of branches. This could be either the history of a single branch or the history of multiple branches in the program. In the case of a multithreaded environment, the situations that we come across are two-fold

1) A multiprogrammed multithreaded model. (SMT)

2) A single programmed multithreaded model. (SpMT)

The thesis uses a multi-core system for single thread applications based on Speculative Multithreading, discussed in chapter 3, to test various branch prediction algorithms. There are differences in the ways branches need to be predicted with good accuracy in a multi-core multithread system and a multi-core single thread system. These differences are in terms of making use of the branch behavior in the scope of a single program or multiple programs. This results in the way prediction hardware is allocated and shared between threads. In a single thread sequential system, the branch prediction accuracy is a lot dependent on the way the history of branches is tracked and made use of, both globally and locally. Studies have shown that global branch prediction has a significant role in improving prediction accuracy. In conventional single-threaded processors, the branch history is recorded in program order; the recorded history thus depends only on the program being executed. 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.

In a SpMT processor, by contrast, the multithreading 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. This results in 5 types of recorded branch history:

1) Insufficient history: the recorded history in very few updates.
2) Discontinuous history: the recorded history does not include some of the updates.
3) Outdated history: the recorded history does not contain the latest updates.
4) Scrambled history: the recorded history is updated in an incorrect order.
5) 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. These issues that we encounter in speculative multithreading prompt us to think of the possible configurations of branch predictors to get around the problems encountered above. In particular, one can think of the following two options:

1) Private predictors
2) 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 tends 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.

1.4.2 Domain shift – From a program to a thread

In SpMT, the domain of branch prediction shifts from a full fledged application to a thread. In systems where thread hopping across the cores is much more frequent thanks to the variations in control independence, the thread sizes tend to be smaller. Short threads do not give the branch predictors enough time to build up good enough history to make the right predictions. 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. Control flow inside a thread, inter thread communication and thread execution style have bearing on the branch prediction inside threads. The way in which threads are forked affects how branches could be updated and hence on how the predictors are distributed across cores. For a highly speculative system where there is a high chance of thread squashing, predictor tables need to be rolled back on wrong thread executions. When updates are not done in program order, it becomes very difficult to do these setbacks correctly.

1.4.3 Identifying the objectives

From the analysis of problems associated with branch prediction when the paradigm shift happens from sequential to speculative execution, we reach a place where we know what could improve conditional branch prediction in SpMT systems.

1) Alleviate the effect of problems brought in by the discontinuity in global history as threads are forked.
2) Provide mechanisms to improve the early predictions within a thread, so that the short thread lengths do not set a bar on the training time of the predictors in each core.

3) Try to make use of the information available from the bigger domain of the entire program and translate or adapt it to the smaller domain of the threads to take care of problems 1 and 2.

4) Make sure that the solution is scalable so that it fits into systems with both high and low number of threads and core hopping.

1.5 Contributions of the thesis

The thesis proposes an entirely new approach to confront the degradation of prediction accuracy in SpMT processors. Previous research done in this field have dealt mostly with methods that are specific to each thread. This thesis proposes a highly scalable solution, by combining both the global (across the whole program) and local (across the threads) behavior of branches. The thesis also proposes a new timing model to effectively simulate a multi-core system by using only one core instance. The dynamic allocation-deallocation of time based updates on a single value (section 5.6.1) successfully emulates parallel updates in time. This is a more efficient and a faster approach for doing performance analysis for multi-core architectures.

1.6 Thesis Outline

The thesis is organized as follows:

Chapter 1 gives an introduction to Branch Prediction and Speculative Multithreading and how prediction differs in the context of SpMT

Chapter 2 looks at the prior work done in this field, their contributions, drawbacks and comparisons.

Chapter 3 discusses Disjoint Out-of-Order Execution (DOE) architecture, the basic framework for the thesis, for which branch prediction algorithms were studied. Chapter 4 explains the simulation methodology adopted for the thesis.

Chapter 5 presents the results.

Chapter 6 proposes the future work along with the conclusions.

Chapter 2

Related Work

This chapter will discuss and highlight the work done towards improving prediction accuracy for multithreaded architectures. Speculative multithreading systems have mostly been at the experimental phase since the time the concept was proposed to improve the performance of single thread applications. Many aspects of SpMT architectures are still under research and branch prediction is one amongst them. The previous work on branch prediction algorithms for SpMT was not widely available in forms of proven and implemented versions with realistic results. Most of the available literature dealt with possible solutions and experimental results based on simulator performance analysis. Most of the earlier work done was for handling branch predictions on sudden context switches. This approach was more inspired by the Simultaneous Multithreading (SMT) [26] architectures where thread switching happened between multiple programs. In this chapter the various studies undertaken in the direction of branch prediction for multi-threading will be discussed and at the end the contributions and shortcomings of these will be summarized.

2.1 RAS and BTB optimizations for multithreaded architectures

The approach that has been used in many of the experiments done so far is to adopt the best strategy that works for single thread performance and use it in the multithreading environment. Previous work in this direction included simulating a multithreaded environment with superscalar cores [5]. Se´bastien Hily & Andre´ Seznec [5] studied and analyzed the performance of the standalone 2 bit bimodal, gshare and gselect predictors on multiprogrammed multithreaded applications (SMT) and single programmed multithreaded applications (which are of interest to this thesis). The focus of their experiments was mainly on measuring the impact of Return address Stack (RAS) and Branch Target Buffer (BTB) [25] implementations on the misprediction rates. The study was also targeted at analyzing the misprediction rates of unconditional branches, which included function calls and returns. RAS is used to predict the return address while coming out of a function in the program flow. The framework used 12 entry stacks for RAS. The BTB sizes were fixed at a base size of 512 for each thread. While the method used to fork threads is unknown for this study, the benchmarks used for performance analysis were applications from SPLASH2 series. Experiments were done by varying the sizes of BTB and the pattern history tables with different numbers of threads. The study concluded that gshare performs better than both 2 bit bimodal and gselect when multiple threads belonging to the same application are run in parallel. The performance of bimodal predictor deteriorated as the number of threads increased. For all the three predictors, it was found that the predictor sizes needed to scale with the increasing number of threads for achieving decent prediction accuracy. Smaller predictor sizes increased the number of BTB misses thereby aggravating the misprediction rate.

2.2 Extrapolation and Correlation of Branch History

Manoj Franklin et al. [10] proposed how branch history can be recreated by extrapolation and correlation for SpMT architectures with short thread lengths. The simulation framework proposed by this work focuses on Per address predictors (Pag). The study looks at the following aspects of branch predictions first before coming up with the proposed solutions.

1) Varying the thread size.
2) Updating the history tables at branch resolution.
3) Updating the history tables at branch commit.

The initial experiments done by Manoj et al. look separately at shared and private predictor configurations for SpMT systems based on a multiscalar [27] threading algorithm. In a shared predictor configuration, a single branch predictor is shared globally by all the cores. In a private predictor configuration, each core has its own branch predictor. For the shared predictor configuration, the performance of a bimodal counter, a per address predictor and a global gshare predictor was evaluated. The observations were as follows.

1) For the bimodal predictor, the accuracy is hardly affected by outdated history (c.f. section 1.4.1). Also scrambled and inaccurate histories do have a small impact on its performance.
2) For per address predictors the maximum deviation in misprediction rate was from 4% on a single thread system to 20% on a SpMT system.

3) For the global gshare, the misprediction rates were found to be very high for small thread sizes.

From his initial experiments Manoj concluded that branch prediction schemes that depend on pattern-based history tend to suffer the most in multi-threaded systems. For the private predictor experiments, one predictor was assigned to each core and performance analysis was done for each of the three predictors. The bimodal predictor was found to be hardly affected by discontinuous history. The performance of pattern-based Pag scheme, which uses only per-address history, was somewhat affected when the thread size was decreased. Discontinuous history had a bigger impact on Pag, because the history used by Pag is “exact” in some sense, and not an “average”. However, the impact was not as severe as in gshare. When the thread size was increased, private Pags began 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 indicated 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.

The experiments pointed towards correction of branch history in multithreaded systems. The study showed that 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.

2.2.1 The extrapolation technique

The extrapolation technique attempts to reconstruct the history that would be available, had the processor been updating branch history in program order and in 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 2.2.1.1 represents a typical implementation of an extrapolation based Pag predictor.

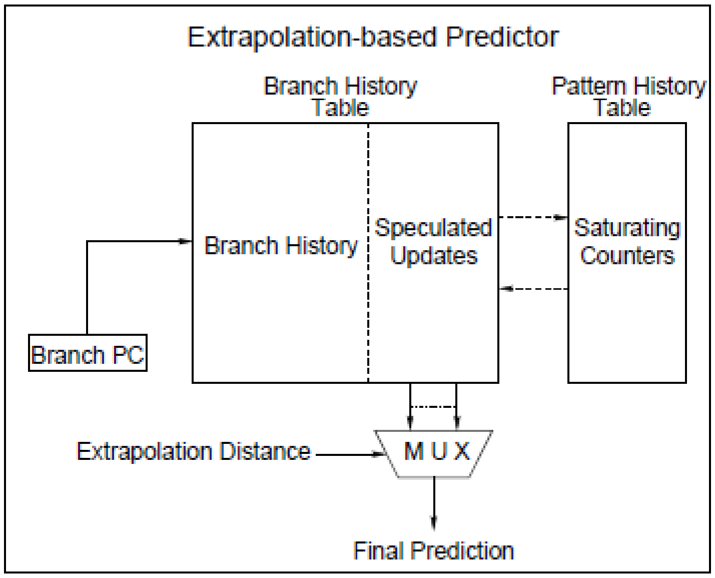

Figure 2.2.1.1 Extrapolation based predictor

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.

2.2.2. Correlation with Thread-Level Information to Obtain Accurate History:

The inherent problem with extrapolation scheme was that when branches are fetched out of order it fails. So a new scheme was proposed to correlate the branch outcomes with the control flow prediction by which the threads are forked. The scheme is illustrated in figure 2.2.2.1

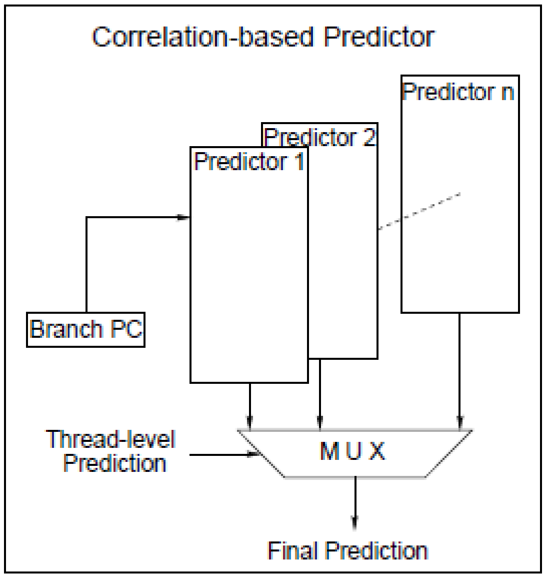

Figure 2.2.2.1 Correlation based predictor

This scheme works on the principle that if 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.

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 worsens the case for some others.

2.3 Per thread history table

The studies done by Manoj or Seznec didn’t look at the dependency of branches on the global history. They also did not solve the problems encountered with the discontinuity in global history in a Speculative Multithreaded environment. The studies and experiments done by Iwama et al. [19] was probably the first step towards addressing this issue. Their analysis looks at a very basic implementation of effective branch predictors that includes the hybrid predictors used for single thread applications and how they can be used to get around the discontinuities associated with global history. Iwama used the bimodal, a per address, a global and a hybrid predictor for the experiments

After initial experiments, Iwama et al. noted that global predictor suffers the most, losing its accuracy by 6.6%. Per-address and hybrid predictor suffer moderate performance degradation of 3.2%, while the accuracy loss of bimodal predictor is only 0.5%.

Iwama attributed his observations to the following reasons:

1) Increasing time to train counters.

   This is the only factor that contributes to the performance loss of the bimodal predictor. Since this loss is almost negligible compared to those of the other predictors, we can deduce that the increased training time is not the limiting factor for performance in gshare, per-address or hybrid predictor.

2) Increasing time to record repetitive patterns.

   To record a repetitive pattern of a loop control branch, it takes N times more iterations for speculative multithreading with N cores than for one core carrying out single threaded execution. This affects prediction accuracy at the beginning of execution and increases compulsory mispredictions.

3) Incomplete branch history.

The branch history in a predictor which belongs to a core does not contain the results of branches executed in other cores. This prevents a predictor from exploiting correct correlation information and leads to poor performance, as shown by per-address, gshare and hybrid predictors.

4) Global history inaccessibility.

Since one thread does not have access to the history register of other threads, a gshare predictor cannot use recent branches' history from the directly preceding thread. It can only use history of branches from previous thread executed in the same core. This explains why a gshare predictor suffers more severely than a per-address predictor.

After observing the above, the direction taken was to confront the discontinuities in branch history due to the timing and spatial differences of consecutive threads being executed in parallel in a SpMT system. The approach taken was to narrow down scope of the program flow to a thread from the entire program. Since this prediction scheme exploited the locality of branch correlation inside a thread, this scheme was called per-thread branch prediction. The scheme works as follows: During the execution of a thread, the history of branch directions are recorded in a register local to the thread, and is used to predict branches in an identical way to the global prediction. When the execution is finished, this history is saved to a table accessible by all threads. This saved history will be used to initialize the history register when the same thread is executed again in the future. To solve the problem when multiple instances of the same thread execute simultaneously, the history was read from the last committed thread. The per thread branch predictor is shown in figure 2.3.1

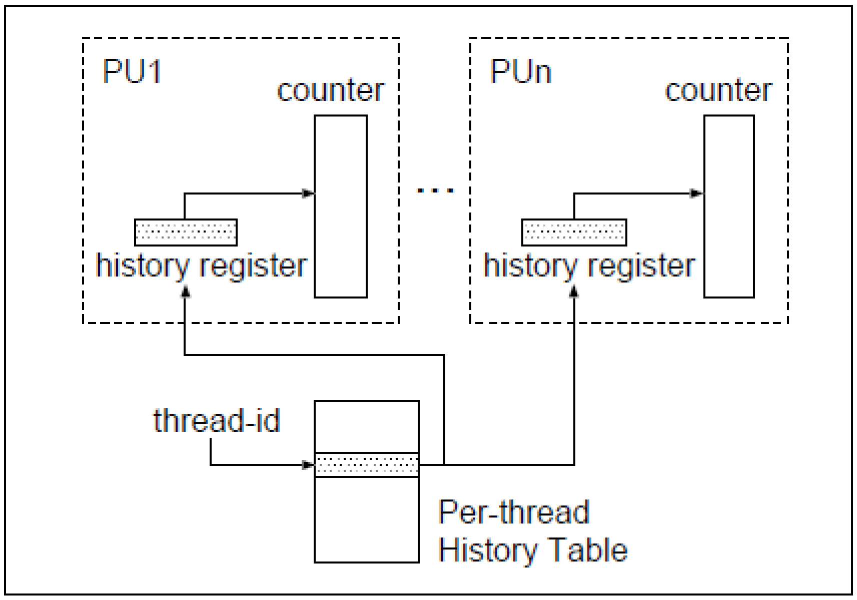

Figure 2.3.1 Per thread predictor by Iwama

According to the results reported by Iwama, the per-thread predictor improved the performance of the gshare predictor by an average of 4.9%. One major setback for the per address predictor alone was that it didn't perform any better than a hybrid of gshare and per address predictor. So Iwama combined the per thread predictor with per address predictors for each thread. The hybrid of per-thread and per-address predictor was more effective than a per-thread predictor of the same size. It outperformed the original hybrid of gshare and per-address predictor for many applications. By using the per thread prediction scheme, the prediction accuracy for go, gcc, ijpeg, and vortex was improved by 1.7% - 2.9%. On an average, the hybrid of per-thread and per-address predictor was the best performing predictor for the speculative multithreading architecture, outperforming the original hybrid predictor by 0.7%.

2.4 Initializing the global history with a consistent starting point

The research done by Dean Tullsen [4] was found to be the most important and pertinent to this research work. Tullsen's focus was on Speculative Multithreading architectures for short threads. His research pointed to the dependency of the prediction accuracy on the correct Global History Register (GHR) value .The observation was that for threads of size 2000 instructions or less, the branch misprediction rate increases, becoming quite significant as the thread sizes get below 500 instructions. 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 provide the exact GHR value, a new core makes use of the residual information from the last thread which executed on that core. This residual data in the GHR typically does not provide any relevant information for 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. Tullsen proposed that his experiments were applicable to any system with frequent thread spawning if not exactly matching the specific SpMT algorithm used in his setup.

The simulation architecture used by Tullsen had little compiler support, to conform more to a dynamic system that made use of hardware structures to train the predictors. For each core simulated, a 2BCgskew predictor [13] was used.

2.4.1 Generating Global History

To improve prediction accuracy and reduce destructive behavior caused by meaningless GHR values, several approaches were considered. These 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 past information. The second category of techniques only seeks to provide a consistent starting point for the branch predictor every time a thread starts up. All techniques were compared with the original setup that retains whatever value was left in the GHR by the last thread which executed on a core.

2.4.1.1 Providing a consistent starting point for the branch history

The idea was to provide a consistent history value for the register, each time a thread started executing. This approach comprised of three schemes.

1) Initialize the history to zero.
2) Use the Program Counter (PC) value of the RIP (c.f. section 3) to initialize the history.
3) Use a XOR of the SP (c.f. section 3) and RIP to initialize the branch history.

Tullsen reported improvement numbers of 38% for misprediction rates and 11% for IPC speedup with the PC initialization scheme. He proposed that the PC scheme could even outperform an actual history (according to single thread flow), if the RIP is chosen such that there is not much of correlation between branches after RIP and the ones before it, by taking away unwanted history which induces noise in the history. He concluded that for threads whose control flow behavior is relatively independent of the pre-RIP branch history, using a single value is at least as good as using the real history. On the other hand, for those threads whose control flow is highly correlated to the pre-RIP history, a manufactured history that uses obsolete data is likely to do more harm than good.

2.5 Summary and comparisons

The experiments done by Hily and Seznec were some of the first in the direction of improving branch prediction for multithreading architectures. Hily and Seznec worked on SMT based systems and not SpMT based systems that are different in the very way threads are created. One basic and major difference between SMT and SpMT is that in SMT the threads belong to different programs and in SpMT they belong to a single program. This affects how the branch behavior can be exploited over the scope of the thread relative to the entire program. Although some of the facts on context switching can be studied and understood from the studies of Hily and Seznec, their applicability to the current thesis is limited in terms of both improvement and scalability. Hily and Seznec propose BTB and RAS optimizations to improve the prediction accuracy. This approach needs scaling in the sizes of both the structures mentioned, as the thread size or the thread numbers increases. (as concluded in the studies). Also, their study doesn’t take into consideration any aspect of history based prediction to optimize and improve prediction accuracy.

The studies and experiments done by Manoj et al. (c.f. section 2.2) dealt with the problems encountered with history in a thread switching environment. While his studies clearly elucidated the specific problems related to discontinuity in branch history, the experiments were done for systems with very small threads, of sizes in 100s, whereas the architecture that this thesis uses is a more realistic model with thread lengths in the vicinity of 1000. Trying to recreate branch history using extrapolation and correlation is not a satisfactory solution in a scenario where the program flow dynamically changes. This fact is aggravated with the not-so simple logic to recreate the branch history as proposed by Manoj. The study itself concludes that extrapolation helps some benchmarks and worsens prediction accuracy in others. Extrapolation works well in a situation where multiple instances of the same branch are fetched in a sequential way. In a realistic multithreading model it is quite possible that multiple instances of the same branch will get executed simultaneously in multiple threads very frequently. Also, the study focuses mostly on per address branch predictors that are more effective while using loop intensive floating point benchmarks where the correlation is more on a per branch basis. Integer benchmarks which resemble the nature of common user applications more, show a global correlation of branches. This makes global history a very important aspect to consider while designing branch predictors for multi-core architectures. Manoj's study doesn’t take into account any aspect relating to global history. One very important contribution that his studies made to this thesis is the fact that the discontinuities in history don’t affect the performance of bimodal predictors a lot. This fact has been made use of in designing the branch predictor for DOE architecture (c.f. section 3.1).

Iwama et al. proposed the first possible solution specifically for SpMT based multi-core architectures. The previous work done till then had mostly focused on Simultaneous Multithreading and did not deal directly and aptly with the branch behavior and issues specific to SpMT. The studies done by Iwama et al. directly addressed the global history issue faced by context switching in SpMT and proposed a novel method of initializing the global history each time a thread starts running on a core. Also, Iwama's proposals were one of the first to use the standard hybrid predictors instead of standalone predictors. The inherent flaw with Iwama's proposals was the usage of the branch history from a previous execution of a thread. This history is stored at the end of the thread execution and is used to initialize the history when the same thread executes again. This has two problems.

1) The stored history corresponds to the end of thread and not the start of the thread.
2) The branch history at the start of a thread will change dynamically during execution.

This means that the history used for initializing the branch history is neither consistent nor thread specific. In fact, during the experimental phase of this thesis, Iwama’s technique was used to initialize the history register for each thread, during the start of execution, by dynamically storing the history as part of the Spawning point (SP) - Control Independent Reconvergence Point (RIP) mapping table. This technique didn’t yield any improvement in prediction accuracy.

Dean Tullsen's work of improving prediction accuracy for short threads using initialization of the history register using the PC of the RIP has been the most useful lead in this thesis. His studies use a specific consistent value corresponding to a thread to initialize the history and reported improved misprediction rates. One point to note here is that this algorithm works best when the branches before and after the RIP are not correlated highly with each other. The improvements in misprediction rates in Tullsen's experiments indicate that the thread spawning algorithm that he used in his experiments was at places like procedure boundaries where the correlation of branches before and after RIP is less. But in a realistic SpMT framework it is not necessary that the threads should start at points with less branch correlation. The methods by which control flow is broken down into threads can be quite different from one SpMT framework to other and the technique of initializing the history register with a consistent value can only aid those threads which are spawned at points with less correlation. For this thesis, Tullsen's strategy of initializing the global history on each core when the thread starts running, was adopted.

Chapter 3

Branch Prediction for Disjoint Out of Order Speculative Multithreaded Architecture

This chapter will discuss the development of the branch prediction framework for the Disjoint Out of Order Architecture.

3.1 Disjoint Out of Order Architecture

Disjoint Out of Order Architecture (DOE) [15] is a multi-core architecture based on Speculative Multithreading for improving the performance of single thread applications. The basic DOE architecture is presented in Figure 3.1.1

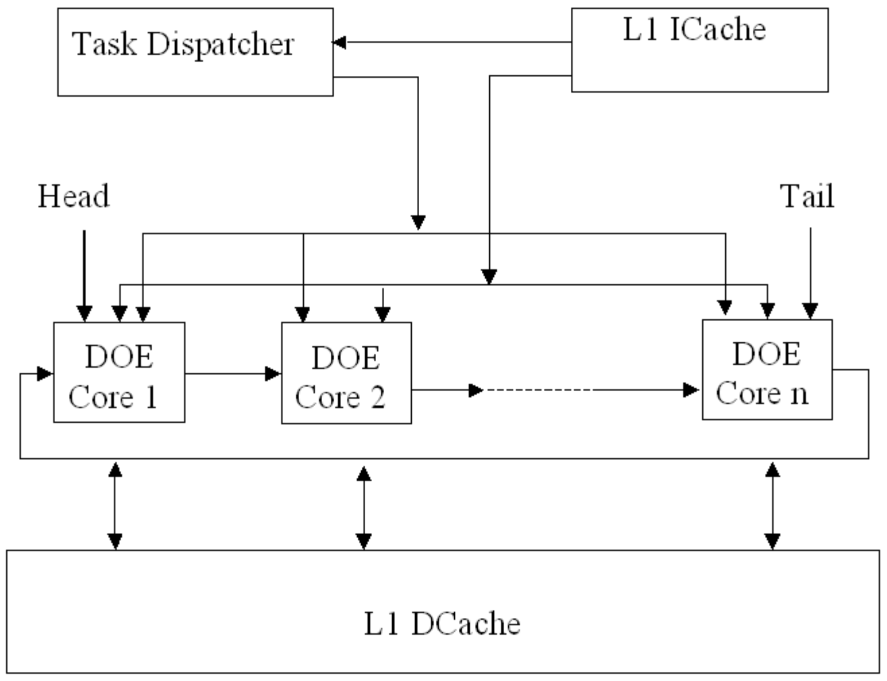

Figure 3.1.1 DOE Architecture

The architecture consists of a collection of cores connected in a ring network. The task dispatcher logic monitors the execution sequence in each core and assigns tasks to the available cores. A task (thread) is a set of contiguous instructions in the dynamic instruction stream. A core fetches and executes the task it is assigned until the end of task. The end of a task is the start of the next dispatched task since tasks are spawned in program order. Although dispatching tasks out of order is possible [3], dispatching tasks in order simplifies data communication since two consecutive tasks run on two adjacent cores in the ring. The tasks are selected to start at future control independent points (RIP) in the program (c.f. section 3). This ensures that regardless of branch execution within a task, the task will ultimately join the next dispatched task.

The execution region within the dynamic instruction trace at a given time is bound by the first fetched instruction in the oldest task and by the last fetched instruction in the youngest task. However, instructions within the execution region are fetched, executed, and retired out of order. The program order of tasks matches the cores physical order in the ring. A head pointer and a tail pointer rotate around the ring. A new task is allocated a core at the tail of the ring, if that core is free. A task at the head commits after it executes all its instructions and reaches (i.e. joins) the first instruction of the next task. When a task commits, its assigned core is freed.

3.1.1 Control independent execution

Each task is control independent of previous branches. DOE performs recovery from mispredicted branches locally within a core without squashing other tasks. This allows the architecture to exploit distant ILP far beyond mispredicted branches, unlike conventional processors which are limited to the amount of ILP available between mispredicted branches.

3.1.2 Disjoint data threads execution

Each core executes two disjoint data threads out-of-order. One data thread consists of all instructions that are data dependent on previous active tasks, and the other data thread consists of all instructions that are data independent of previous active tasks. The independent data thread starts execution immediately when a task is dispatched, while the dependent data thread is buffered outside the execution pipeline and executes when the previous task completes and commits. By this time all previous mispredicted branches have been corrected and input data propagated from the previous task. The dependent data thread therefore does not block the execution of the independent thread. This achieves two goals: 1) it supports control independent execution of tasks, and 2) it provides tolerance to the delays encountered on input data produced by other cores and communicated through the ring.

3.1.3 Checkpoint processing and recovery

By using checkpoints for recovery [14],[15], completed independent instructions can pseudo-retire [15], freeing the pipeline resources. Two checkpoints are taken for any speculative thread, one at the beginning of the thread and the other at the end of the dependent instruction execution. In the case of an exception or misprediction in the dependent data thread, DOE restores precise state from the first checkpoint and restarts execution after flushing the pipeline and squashing the task. If the data dependent thread completes without exceptions or branch mispredictions, DOE integrates the results from the dependent and independent threads by merging states from two different checkpoints.

3.2 DOE Core Microarchitecture

Figure 3.2.1 shows a block diagram of the DOE core microarchitecture.

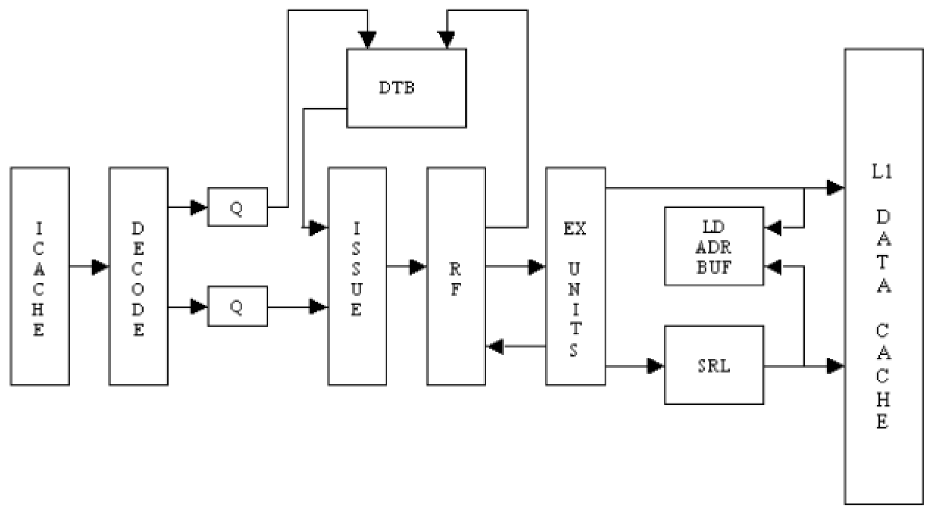

Figure 3.2.1 DOE Core Micro Architecture

3.2.1 Dependent Thread Buffer (DTB)

The DOE core handles dependent instructions differently from independent instructions. When a task is spawned, a bit mask that identifies the set of influence registers (also called live-in registers) is loaded from the task predictor into a poison bit vector located in the decoder block of the core that is assigned to execute the spawned task. The influence registers are the ones which create data dependency between threads. The dependents of all live-in influence registers and their descendents are extracted from the task instruction stream using the poison bit vector in the pipeline decode stage and are stored in a Dependent Thread Buffer (DTB) outside the pipeline. The dependent thread instructions therefore do not consume or occupy precious pipeline resources such as issue and buffer slots or pipeline staging latches. This completely frees the pipeline resources for independent instructions to execute. Since dependent instructions do not tie pipeline resources, the core achieves a continual flow of execution [15] and can look ahead far into the task for independent instructions to process, until the task reaches the end or until the previous thread completes. When the previous task completes, a register checkpoint is taken and execution switches to the dependent instructions in the DTB. When all the DTB instructions execute, their results are merged with the independent instruction results from the checkpoint. The checkpoint is then discarded and execution resumes normally without having to go back to execute independent instructions again.

3.3 Branch Prediction for Disjoint Out of Order Architecture

The thesis focuses on conditional branch prediction for DOE. Since the threads are control independent of each other, the branch predictions are handled within each thread for DOE. The conditional branches in DOE can be divided into two categories.

1) Independent conditional branches.

2) Dependent or poisoned conditional branches.

The independent conditional branches are the normal branches that are encountered in a thread which are not dependent on any other thread through memory or registers. This means that the source of the branch instruction is not produced by any of the previous threads in the ring. They behave normally in each thread, as a branch would in the sequential execution, with the domain changing from the whole program to a thread. Whenever a branch misprediction occurs, the pipeline is flushed and the fetching starts from the right path. The second category of branches, which are poisoned, get drained into the DTB. These branches don't commit, and reside in the DTB till the previous task finishes and then they execute out of the DTB. Once the poisoned branches re-issue from the DTB, they are predicted and if the prediction is right, then execution proceeds normally. If there is a misprediction, the processor rolls back to the last checkpoint which is the start of the thread. This makes the poisoned branches much crucial for DOE since the poisoned branch mispredictions can result in thread squashes and restart which is a heavy toll on both power and cycle time.

3.3.1 Initial framework and evolution

The early thought process of branch prediction for DOE was similar to that of a Simultaneous Multithreaded System. The basic implementation was a McFarling Configuration [2] of Bimodal and Gshare Predictors to see how branch prediction gets affected when the domain changes from a purely sequential execution to DOE mode. The initial implementations were a shared predictor for all the cores and a private predictor for each core. These frameworks gave results with the shared predictor performing worse than the private predictors as corroborated by earlier studies. The shared predictor being subject to out of order updates from threads was not able to provide consistently correct predictions. This resulted in degraded prediction accuracy. So a private predictor scheme, where each core has its own branch predictor, was fixed as the baseline configuration for the experiments.

Confronting the discontinuities in history was the next step in moving ahead. Previous studies [3],[4] showed that providing a consistent starting point for the Global branch History Register (GHR) at the start of each thread was the most optimal way to reduce mispredictions in the absence of correct history. The best results with this setup were reported in the study by Tullsen et al. [4]. This scheme was adopted as an initial step to reduce mispredictions due to lack of proper history at the start of a thread. Since this scheme uses the address of the instruction which reconverges (c.f. section 3) we call this scheme the Reconvergence IP (RIP) scheme. Figure 3.3.1.1 depicts this scheme

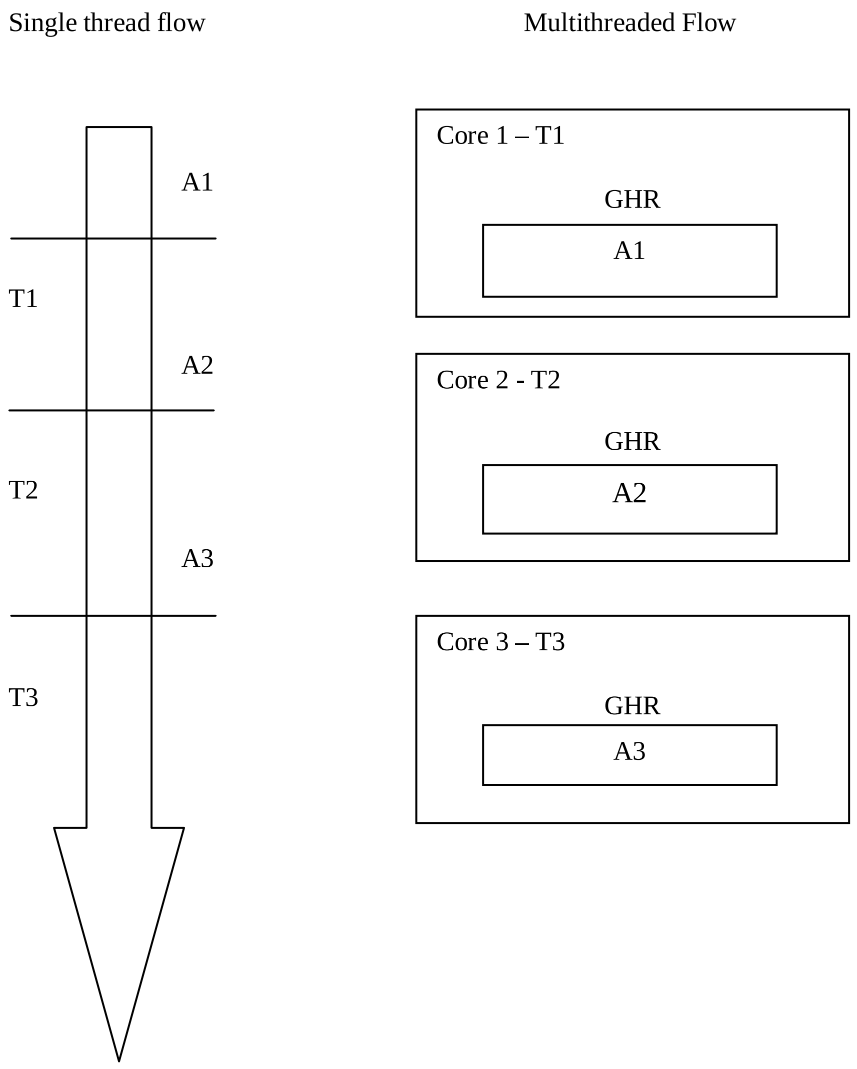

Figure 3.3.1.1 RIP Initialization Scheme

3.3.2 The Aliasing Problem and choice of 2BCgskew as the base predictor

Traditional branch predictors (which include Gshare and Per address) which hash into the predictor tables with vectors derived from the address bits and the history, suffer from a problem called aliasing. Aliasing occurs when two branches with completely different directions point to the same Pattern History Table (PHT) [2] entry. This happens when the resultant vectors from combining the lower address bits and history bits for the two branches are the same. The problem of aliasing has plagued branch predictors since the development of the basic predictors. New schemes were developed to solve this. Aliasing takes a more dangerous dimension in DOE since the space for recovery from aliasing is narrowed down from the complete program to that of a thread. This effectively leads to higher chances of misprediction which are costlier in DOE. The 2BCgskew predictor used in the legendary Alpha Ev8 processor by DEC [13] has been proven one of the most efficient de-aliased predictors for single thread applications. The basic principle behind de-aliasing is using multiple indexing schemes with the same address and global history. If aliasing occurs in one indexing function, it may not occur in the other one even though the sources of both combinations are same. Aliasing recovery can be quite time consuming and in small threads the training time is very less for the predictor tables.

Figure 3.3.2.1 shows the architecture of a 2BCgskew predictor.

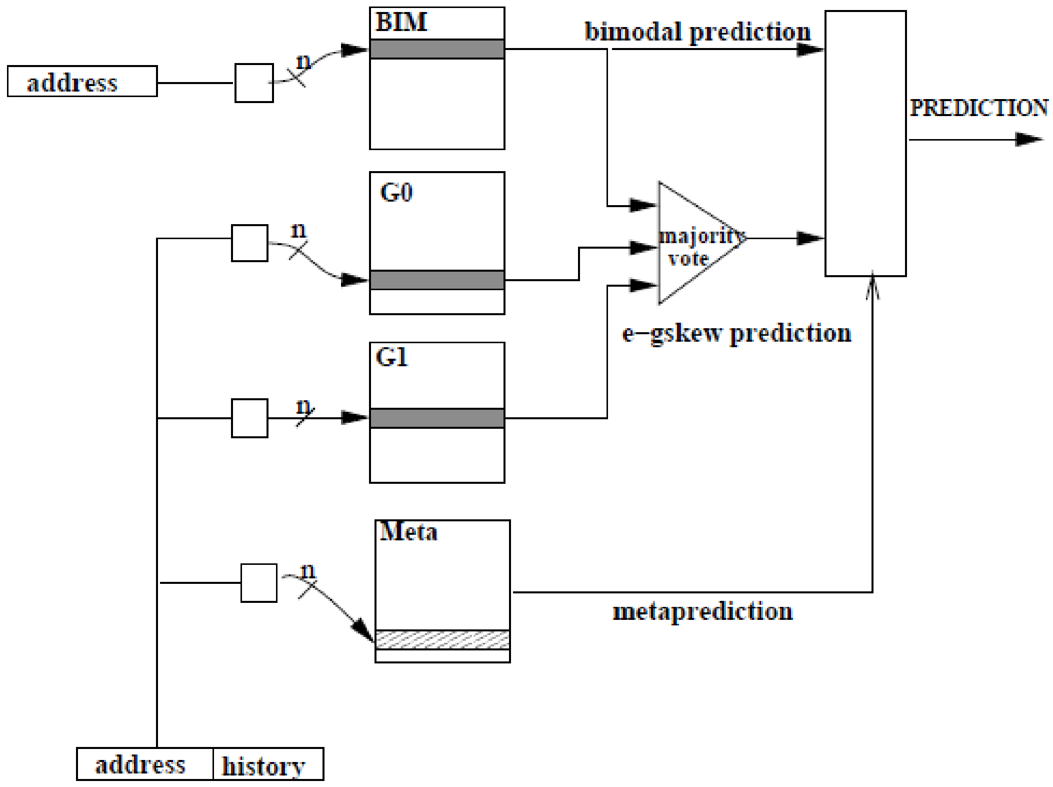

Figure 3.3.2.1 2BCgskew Predictor

2Bcgskew consists of four 2-bit counters banks. Bank BIM is the bimodal predictor, but is also part of the e-gskew predictor [22]. Banks G0 and G1 are the two other banks of the e-gskew predictor. Bank Meta is the meta-predictor. The meta predictor chooses between the prediction from BIM and the majority vote on the predictions from G0, G1 and BIM. The bimodal component accurately predicts strongly biased static branches. Therefore, once the metapredictor has recognized this situation, the other tables are not updated and do not suffer from aliasing associated with easy-to-predict branches.

3.3.3 Broadcast mechanism for DOE with Global Broadcast Buffer

So far the focus has just been on how well the history can be modeled to emulate a sequential flow for SpMT systems. When the whole program in broken down into small threads in SpMT, the degree of freedom with respect to finding interactions between branches, which is crucial in updating the prediction counters, becomes very less. The training time for the predictors to reach a state where they can make correct predictions can be a considerable part of the total execution time of the thread. Any branch prediction scheme with heavy dependency on the history will suffer badly when plugged into this situation. In the context of DOE this becomes more crucial since we deal with thread sizes of 600 to 800 on an average. So the problem becomes twofold: a) How to reduce aliasing which could be really crucial for small threads b) How to provide a stable prediction mechanism so that the predictors need the least time to warm up or can make predictions correctly independent of the thread size. The broadcast scheme answers these two questions effectively.

3.3.3.1 How to reduce aliasing in a smaller thread domain with minimal dependency on history and minimal counters: Bi-mode to YAGS

One of the most remarkable features of branches in a program is that a considerable percentage of the total branches in the program are biased in one direction, either taken or not taken. According to the variations in history of the prior branches, sometimes the branches can go in a way that is opposite from their biased direction. This dependence of branches on branch history is unquestionable and can't be ignored in any design. At the same time the aliasing problem that cripples the history indexed predictors have to be looked at. A simple solution which combines these aspects of the branches is the YAGS scheme [7]. YAGS is an improved version of the bimode predictor [6]. The following discussion will focus on how a bimode predictor effectively reduces aliasing for gshare schemes using simple mechanisms. Figure 3.3.3.1 depicts a bimode predictor.

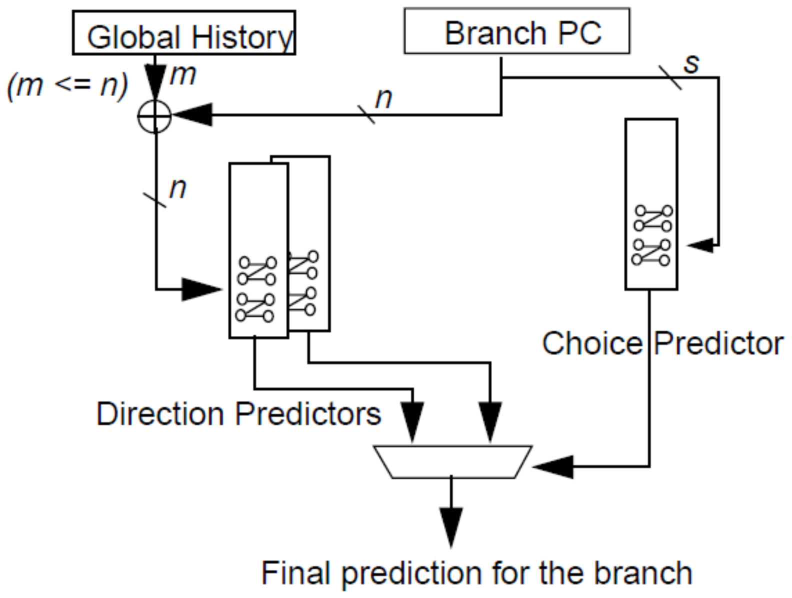

Figure 3.3.3.1 Bi-mode predictor

In the bi-mode predictor, the saturating counter Pattern History tables (PHT), indexed by the XORing of global history and branch address, are divided into two tables. Each of these is called a direction predictor. Either of these is chosen by another top level predictor which is called the choice predictor. The choice predictor is essentially a bimodal predictor indexed by the lower order address bits of the branch. Each branch has essentially a biased behavior. So whenever a branch is encountered, the choice predictor will choose the direction predictor corresponding to the biased behavior of the branch. This means that the direction predictor in the taken direction of the choice predictor will mostly be predicting "taken" as the direction, and same is the case with not taken branches. If two branches get aliased thanks to the resultant index pattern, still the choice predictor will choose the direction predictor in the biased direction of the branch. This helps the predictor to reduce aliasing effectively. When the direction of the branches takes the opposite direction to its biased direction thanks to a specific history, it will get updated at the corresponding direction predictor. Thus the limited number of instances where the branch deviates from its usual behavior is taken care of in the direction predictors. This behavior suggested by the bi-mode predictor, to effectively reduce aliasing, is quite suited for a framework like DOE with short threads and frequent context switching.

The motivation behind YAGS is the observation that for each branch we need to store its bias and the instances when it does not agree with its bias. If we use a bimodal predictor to store the bias, as the choice predictor does in the bi-mode scheme, then all we need to store in the direction PHTs are the instances when the branch does not comply with its bias. This reduces the amount of information stored in the direction PHTs, and therefore the direction PHTs can be smaller than the choice PHT. To identify those instances in the direction PHTs we add small tags (6-8 bits) to each entry, referring to them now as direction caches. These tags store the least significant bits of the branch address and they virtually eliminate aliasing between two consecutive branches. When a branch occurs in the instruction stream, the choice PHT is accessed. If the choice PHT indicated “taken”, the “not taken” cache is accessed to check if it is a special case where the prediction does not agree with the bias. If there is a miss in the “not taken” cache, the choice PHT is used as a prediction. If there is a hit in the “not taken” cache it supplies the prediction. A similar set of actions is taken if the choice PHT indicates “not taken,” but this time the check is done in the “taken” cache. The “not taken” cache is updated if a prediction from it was used. It is also updated if the choice PHT is indicating “taken” and the branch outcome was “not taken.” The same happens with the “taken” cache.

Figure 3.3.3.2 depicts a YAGS predictor.

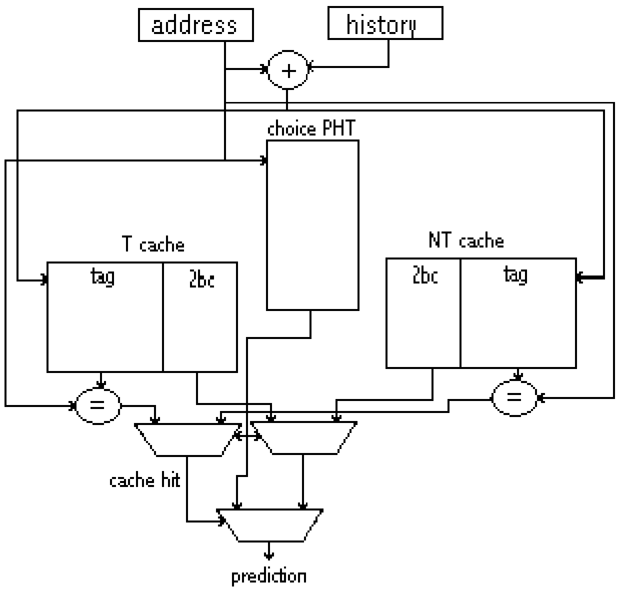

Figure 3.3.3.2 YAGS Predictor

The aspects of the YAGS predictor that help reduce aliasing and make decisions based on the biased nature of the branches made it the final choice for the framework for the broadcast scheme for DOE.

3.3.3.2 How to reduce the warm up time for the predictors to start making the right predictions in the absence of a long sequence of instructions – Boosting the choice bias by broadcast

The answer to the above question lies in the broadcast scheme. The underlying principle of the broadcast scheme is to exploit the interaction between cores in helping each core make the right decision. Broadcast mechanism has three key features.

a) Boost the biased nature of the branches by broadcasting updates as each branch in each core updates its PHT entry.

b) Selectively sample the speculative updates made by other cores so that the initial decision made by the choice predictors in each core, as the thread starts running, is more accurate.

c) Keep the local updates in each core to make decisions in the thread's domain.

These features will be explained in the sections below. Figure 3.3.3.3 depicts the broadcast framework for a multi-core architecture with two cores. Each core uses a YAGS predictor as the base predictor in the broadcast scheme. There is a Global Broadcast Buffer (GBB) which is a pattern history table (PHT) of bimodal counters indexed by the branch address. Each core has a choice predictor which is a bimodal predictor. As each local choice predictor is updated, the result is broadcast speculatively into the GBB. The corresponding entry in the GBB is updated with multiple broadcasts from multiple threads. Whenever a thread starts executing in a core, the history register in that core is initialized with the starting address of the thread (according to the RIP scheme). For the choice predictor, each local choice predictor PHT entry is associated with a local update bit. This bit is reset to zero at the start of each thread. For every branch address, if the local update bit is zero, the choice predictor reads the PHT entry from the GBB. When the corresponding branch is committed, local as well as the GBB PHT entry is updated.

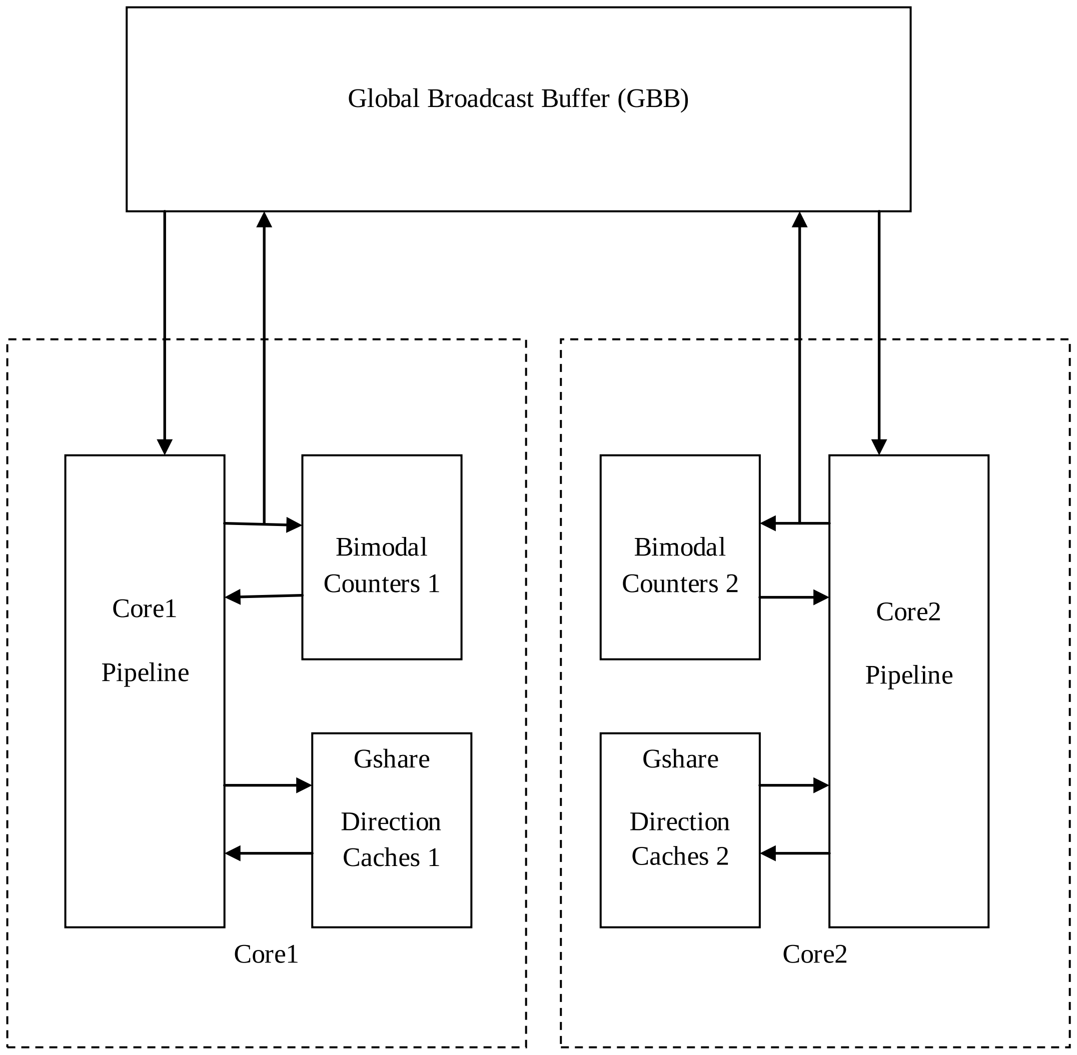

Figure 3.3.3.3 The broadcast scheme with GBB

Once a branch updates the local PHT, the local update bit corresponding to that entry is set. So the next prediction of the branch happens from the local PHT itself. Figure 3.3.3.4 illustrates how one PHT entry is read and updated in the broadcast scheme. The broadcast scheme provides satisfactory answers to the two questions raised at the start of this section. By buffering the broadcast updates from the local choice predictors over time, the global buffer reinforces the biased nature of branches which remains invariant irrespective of the point of execution of the branch in the entire program. This value when read initially at the start of each thread, potentially gives a good prediction for a branch that is executed for the first time in a thread out of order with other threads. The resident values in the direction predictors will mostly reflect the prediction based on the history available, a decent approximation provided by the consistent starting point given by the thread start address.

For each branch in a thread, the direction chosen on the very first execution is very crucial. This is provided by the choice predictor with a buffered global read from GBB. The GBB values are sampled at thread boundaries and at the very first prediction of every branch in a thread. This increases the probability of each branch reading a stable value with respect to its bias. At the same time, once the GBB is read, preference is given to the local updates from then on, using the local update bit. This takes care of the local control flow inside each thread.

Figure 3.3.3.4 illustrates the predict and update of one local bimodal counter entry using the GBB scheme.

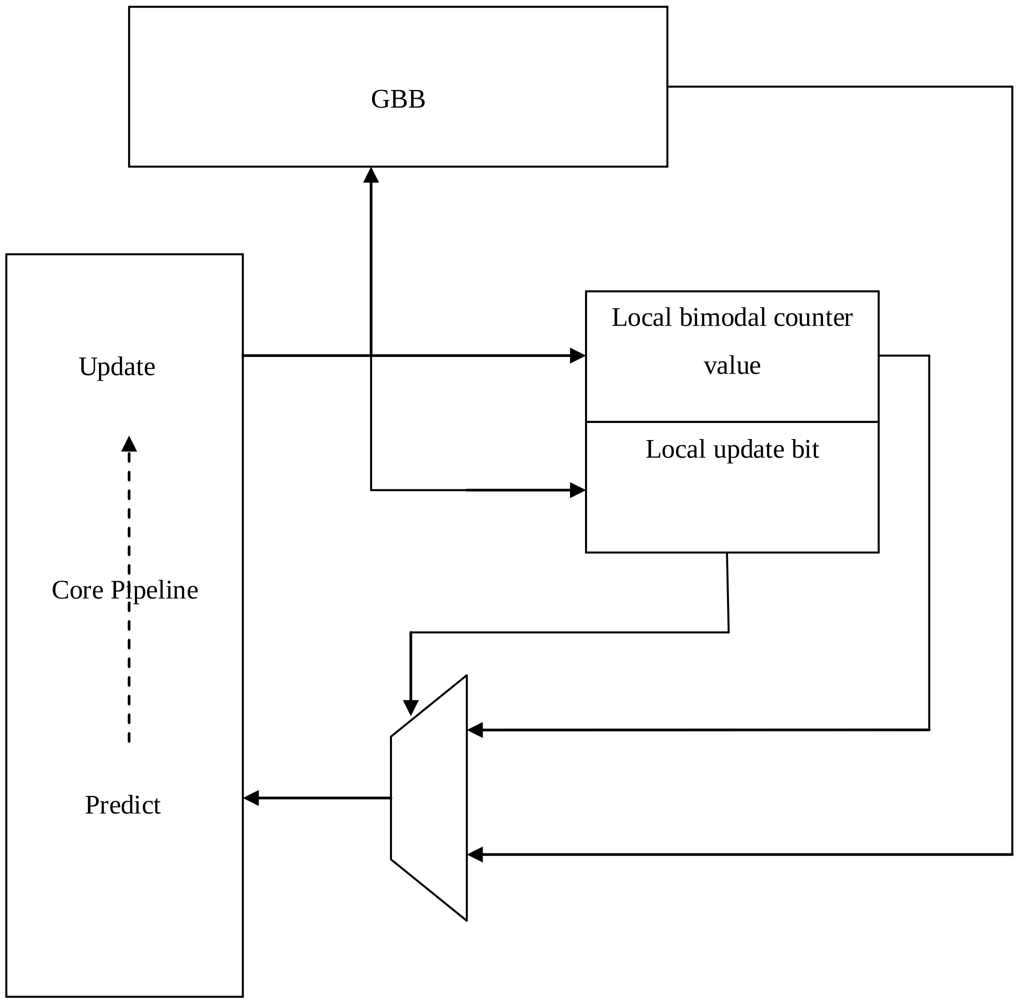

Figure 3.3.3.4 The Predict and Update using the GBB scheme

Chapter 4

Simulation Methodology

4.1 PTLsim - A cycle accurate simulator

The framework chosen for this thesis is PTLsim, a cycle accurate x86 simulator. PTLsim is a state of the art cycle accurate microprocessor simulator and virtual machine for the x86 and x86-64 instruction sets [16]. PTLsim models a modern superscalar out of order x86-64 compatible processor core at a configurable level of detail ranging from full-speed native execution on the host CPU all the way down to RTL level models of all key pipeline structures. In addition, the complete cache hierarchy, memory subsystem and supporting hardware devices are modeled with true cycle accuracy. PTLsim supports the full x86-64 instruction set of the Pentium 4+, Athlon 64 and similar machines with all extensions (x86-64, SSE/SSE2/SSE3, MMX, x87). PTLsim is written in C++ with extensive use of x86 and x86-64 inline assembly code.

4.1.1 PTLSIM – in more detail

PTLsim completely models a modern out of order x86-64 compatible processor, cache hierarchy and key devices with true cycle accurate simulation. The basic microarchitecture of this model is a combination of design features from the Intel Pentium 4, AMD K8 and Intel Core 2, but incorporates some ideas from IBM Power4/Power5 and Alpha EV8. The simulator directly fetches pre-decoded microoperations but can simulate cache accesses as if x86 instructions were being decoded on fetch. The branch prediction is configurable. PTLsim includes various models including a hybrid g-share based predictor, bimodal predictors, saturating counters, etc. The front end pipeline in PTLsim has configurable number of cycles to simulate x86 decoding or other tasks and this is used for adjusting the branch misprediction penalty. Unlike in some microprocessors, PTLsim does not do speculative scheduling: the schedule and register read loop is assumed to take one cycle. Functional units, mapping of functional units to clusters, issue ports and issue queues and latencies are all configurable. The load and store queues use partial chunk address matching and store merging for high performance and easy circuit implementation.

4.2 Branch Prediction in PTLsim

PTLsim comes with a basic hybrid McFarling [2] predictor. This combines a bimodal predictor and a gshare predictor and a meta predictor which chooses between them. Each table is 64K in size. The branch target buffer (BTB) comes with a size of 4K. The BTB is shared between conditional and indirect branches and is implemented as a 4 way set associative table with 1024 sets and updated using Least Recently Used (LRU) algorithm. Each branch is predicted at the fetch stage and the counters are updated at the commit stage of the pipeline. The different parts of the branch prediction mechanism are implemented using functions with configurable parameters. The basic framework provides the user with a large degree of customizability for implementing different prediction algorithms. The delays or latencies can be modeled at each stage based on the size and location of a specific element in the entire framework.

4.3 Implementing DOE in PTLsim

PTLsim implements a customizable out-of-order x86 pipeline. It can be tuned either as a single thread or a multithread (SMT) model. For customizing PTLsim into a SpMT model, initial experiments were done by instantiating multiple cores and emulating a full fledged multi-core scenario. Studies and initial research into the simulator framework proved that it is easier to use a single core and emulate the DOE multicore execution using a timing model without physically instantiating different cores. This is one important highlight of the thesis, based on which, further schemes including branch prediction were built. The forking algorithm for DOE is based on the Control Independence Prediction used in the Dynamic Multithreading Model [3], where each thread is forked on function calls. There is enough control independence on function calls for the threads to be forked that the main function runs on one thread and the parallel thread or the child thread starts running on a different core from the return point. So the function call becomes the spawning point (SP) and the return becomes the reconvergence point (RIP) (c.f. section 3), where the new thread starts.

4.3.1 Doe_cycle - A multi-core timing model using only a single core instance

The basic timing element of the DOE simulation model is called the doe_cycle. The doe_cycle indicates the simulation time for the parallelized simulations of a single program. The simulator also has sim_cycle which indicates the timing of the purely sequential flow. Sim_cycle increments as an instruction goes through each stage of the pipeline. The doe_cycle increments along with the sim_cycle initially. When a function call is reached in the execution, which is a potential spawning point, the time is noted as the starting point of the speculative thread. As the execution continues, if the execution hits the reconvergence point according to the prediction algorithm, then the doe_cycle is pulled back to the thread starting time noted earlier when the spawning point was encountered. This way a parallel timing corresponding to the DOE architecture is emulated. The same thing happens when the speculative thread spawns another task. Since DOE works in a ring manner, where one thread forks only one task, this timing model suits better without the added overhead of instantiating multiple cores, which saves significantly on simulation run time and memory usage. Thus, in the simulator, all the threads run in a sequential way, but only the timing is modeled so that each thread is associated with a time, had those threads run in parallel. Doe_cycle is used as the basic timing element for the implementation of the broadcast scheme for branch prediction. Figure 4.3.1.1 explains the doe_cycle timing model.

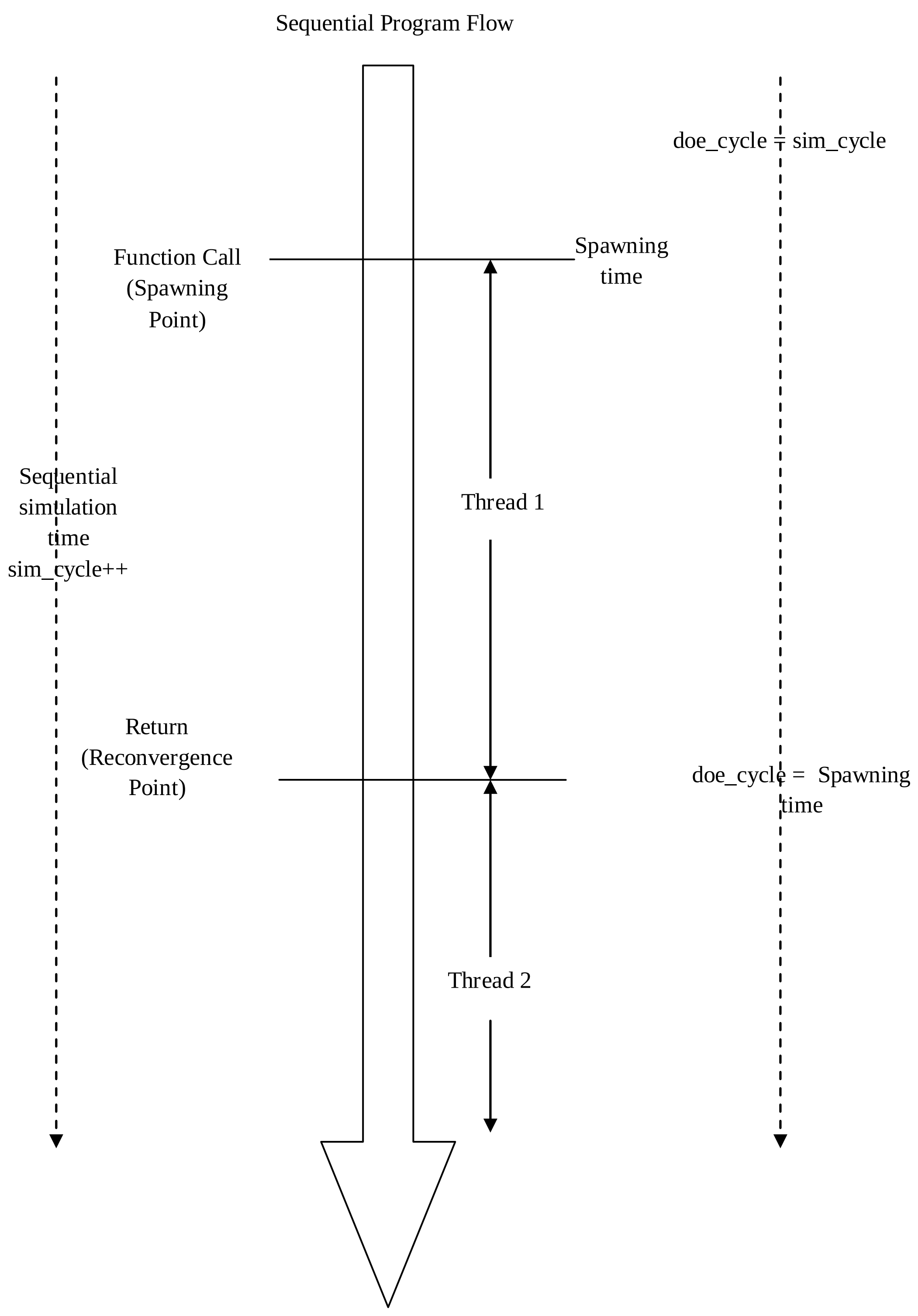
Sequential Program Flow
doe_cycle = sim_cycle
Function Call (Spawning Point)
Spawning time
Sequential simulation time sim_cycle++
Thread 1
Return (Reconvergence Point)
doe_cycle = Spawning time
Thread 2

Figure 4.3.1.1 DOE timing model explained

The fetch mechanism , cache hierarchy and virtual memory mechanisms were kept as if they belonged to a single core. Also the indirect branch predictors and the Return address Stack were not altered. Retaining these parts of the architecture helped in two ways.

1) Keep the simulator complexity simpler.

2) Idealizing the aspects other than the one under study helps in assessing the impact of altering only that aspect, in this case, branch prediction, on the entire architecture.

4.4 SPEC1NT2000 benchmarks

The simulations were run using the integer benchmarks from Standard Performance Evaluation Corporation (SPEC). SPEC CPU2000 focuses on compute intensive performance, which means these benchmarks emphasize the performance of the computer's processor (CPU), the memory architecture and the compilers used. SPEC CPU2000 consists of two subcomponents that focus on two different types of compute intensive performance: CINT2000 for measuring and comparing compute-intensive integer performance and CFP2000 for measuring and comparing compute-intensive floating point performance.

SPEC CPU2000 provides a comparative measure of integer and/or floating point compute intensive performance. These benchmarks are developed from actual end-user applications and not synthetic. Hence the results of simulations done using these benchmarks are much closer to the ones obtained using realistic applications. The integer benchmarks closely resemble the behavior of normal single thread applications and hence were chosen for the research. For the thesis experiments the benchmarks that have a greater sensitivity to branch mispredictions were chosen. These are bzip2, gzip, gap, eon, parser, perlbmk, twolf and vpr.

4.5 Implementing the 2BCgskew predictor and YAGS

PTLsim comes with a very basic branch prediction scheme which is a McFarling predictor with a bimodal and gshare scheme and a choice predictor. This scheme was altered to create the frameworks that are the best to date. The 2BCgskew predictor [13] was chosen as the base predictor configuration and the comparison of the broadcast scheme was done with this scheme. For both schemes under study, the predictor instantiations were replicated to the number according to the number of cores used so that each core has one private predictor. Similarly the YAGS was also replicated for each core. Each table in the 2BCgskew and YAGS were chosen to be of size 64K, conforming to the base sizes that PTLsim came with.

4.6 Implementing Broadcast scheme

The broadcast scheme was implemented using the doe_cycle timing methodology to emulate parallelization using a single core instance (c.f. section 5.3.1). This necessitated emulating a scenario where one value gets updated over time and then the old values in time being read again as the doe_cycle is pulled back to the spawning time. The following example explains this: Thread1 (T1) starts at time 0 and forks Thread2 (T2) at time 100. This means that at time 100 T1 and T2 start running parallely, whereas, in the simulator they run one after the other. As T1 runs, it updates the entries in the GBB. When T2 starts running, it should read the values from the GBB updated before time 100 or the time that any branch of T2 reads from the GBB. This needed storing all the values of the updates to one location in the GBB, as threads run, and allowing subsequent threads to read the values corresponding to time that was before the time of the read in terms of doe_cycle. This was implemented using an array of timestamps and values which were dynamically allocated and de-allocated as the execution progressed. This way, it is possible to emulate simultaneous updates with multiple threads with only a single core instance. The implementation was done with dynamically increasing and decreasing linked list and was a good exercise in understanding memory handling in a simulator like PTLsim. For the GBB a table is instantiated which is accessible to all the threads in addition to the private predictor instantiations. The method is explained in the following section.

4.6.1 Multiple timestamp allocation and de-allocation for emulating multi-threads using single core

Figure 4.6.1.1 explains how updates on a single value by multiple threads is emulated using a single core instance.

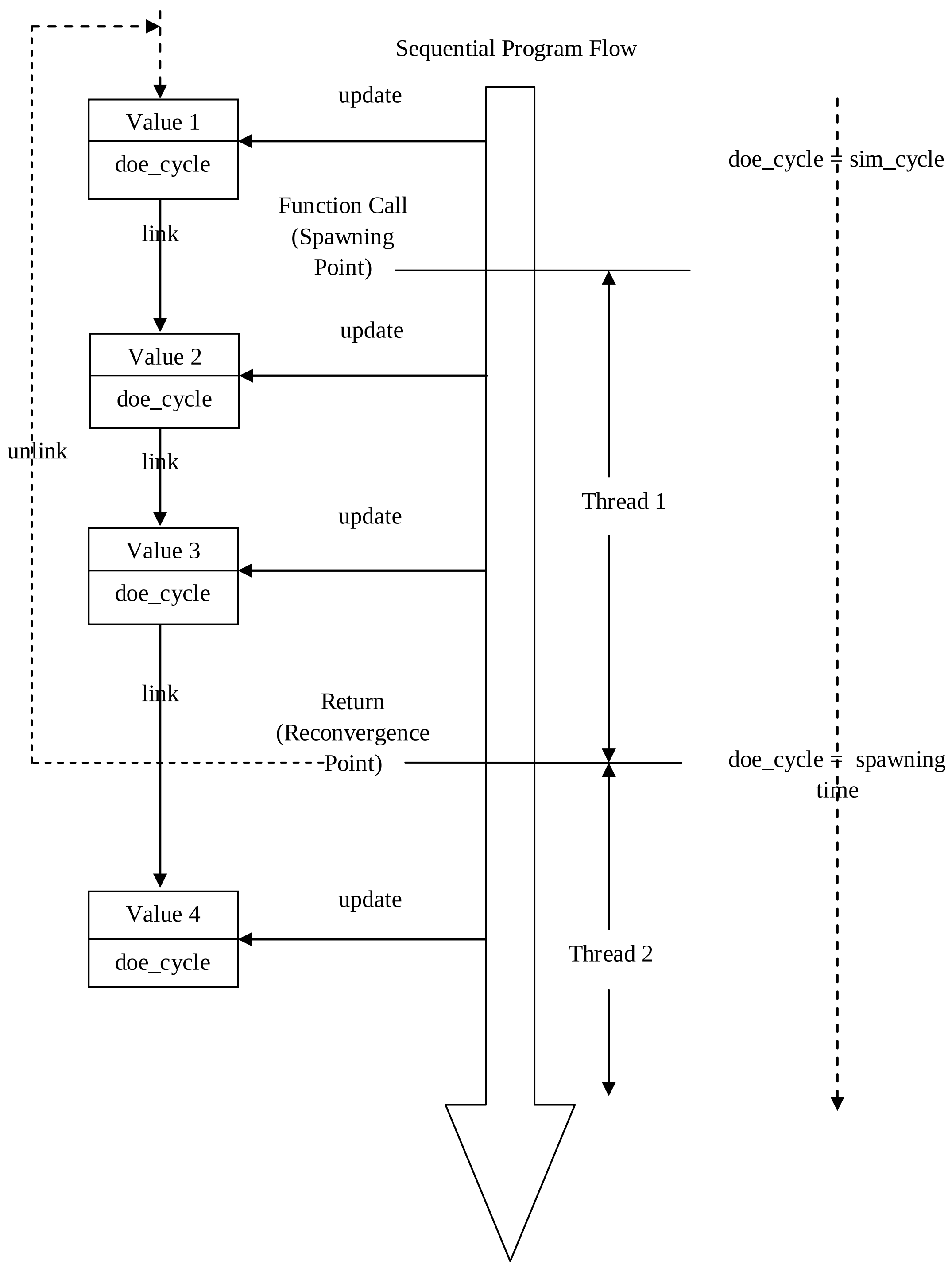

Figure 4.6.1.1 Emulation of parallel updates on a single value using sequential flow

A single value is a base node of a dynamically increasing/decreasing array of values. As the value gets updated at different part of the program a new node is dynamically added (linked) to the base node or the last node added. This happens in the increasing order of doe_cycle. The objective is to preserve the updates of a single value over time. When the sequential execution reaches a point in the program where a reconvergence point (start of the new thread) is encountered, the timing is pulled back to the spawning point using doe_cycle as explained earlier. Also all the values but one before the spawning time is discarded since a new thread just needs only the set of values at the starting point. This dynamic allocation/de-allocation was quite useful in memory handling and also can be applied to other aspects of multi-core architectures like cache hierarchy.

The simulations were run for a trace of 70 million instructions with no warm-up. This means that the branch predictor tables are not trained before they started predicting the directions. Since the core hopping is much more frequent in the SpMT scenario, having a warm-up for the tables per core doesn't make a lot of difference. This situation also suits the realistic situations more. The number of threads was set at 4 for all the simulations.

Chapter 5

Results

This chapter will discuss the results of the simulations with the different predictor configurations that have been studied in the thesis.

5.1 Single thread execution

The decision to choose YAGS as the basic predictor for the DOE started with experiments done at the single thread. The configurations compared are the base McFarling predictor with the bimodal and gshare, 2BCgskew and the YAGS predictor. Figure 5.1.1 shows the comparison of misprediction rates of the three basic configurations

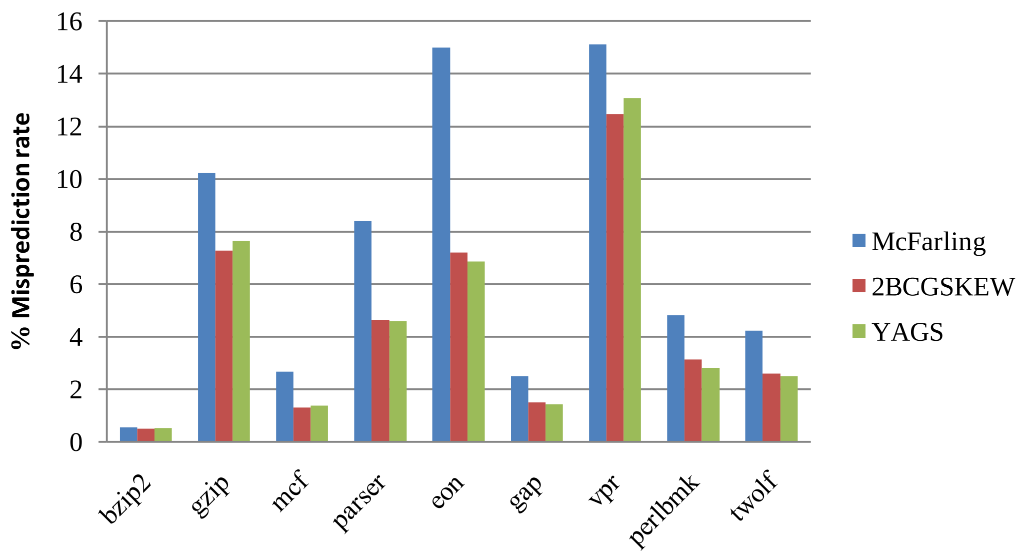

Figure 5.1.1 Single thread misprediction rates

Table 5.1.1 shows the sizes of the tables used for all the three configurations.

Predictor	McFarling	2BCgskew	YAGS
Size	768 (192x4)	1M (256x4)	768(192x4)

Table 5.1.1 Size budget for various schemes

YAGS predictor provided misprediction rates for single thread execution better than both 2BCgskew and the McFarling predictors. The benchmarks gzip, mcf and vpr were showing exception for this behavior by a little margin. Eon showed the greatest improvement. The behavior for mcf, vpr and gzip could be explained by the fact that these benchmarks take advantage of the availability of the available history in the purely single thread sequential execution.

5.2 Disjoint Out of order Execution

5.2.1 DOE worst case

This represents the scenario in which each of the branch predictor per core reads whatever remnant values it has at the start of each thread. There is no optimization or improvement applied to any of the predictor. This was the first step in assessing the impact of multithreading using SpMT on the proven single thread schemes. Figure 5.2.1.1 illustrates the misprediction rates for each predictor in the multithreaded mode.

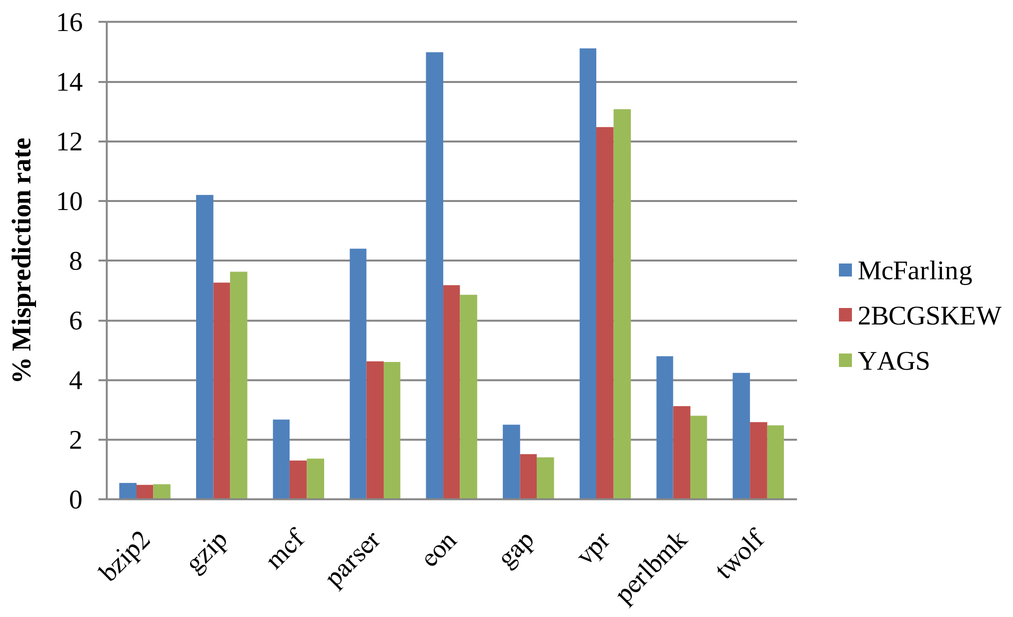

Figure 5.2.1.1 Worst case Multicore Misprediction rates

The results clearly show that the discontinuity in branch history is really taking a toll on the prediction accuracy. The McFarling predictor suffers the most with a huge percentage of degradation in misprediction rate. 2BCgskew performs worse than YAGS for only two benchmarks mcf and vpr. For the rest of the benchmarks YAGS provides the best misprediction rate at 3/4th the size of 2BCgskew.

After the initial experiments, McFarling was not included in the further studies since both 2BCgskew and YAGS were proven to be much more effective in preventing aliasing and achieving higher prediction accuracy. So the base model was fixed at the 2BCgskew (according to the best known scheme in the studies so far) [3]. This scheme will be termed as baseline the further discussions.

5.2.2 DOE with RIP initialization of global history

The starting point of the studies in confronting discontinuities in history was to initialize the global history using a consistent pattern for each thread. This was chosen to be the address of the starting instruction of the thread and is called the RIP scheme. Figures 5.2.2.1 shows the comparison of both 2BCgskew and YAGS schemes with the RIP initialization schemes applied.

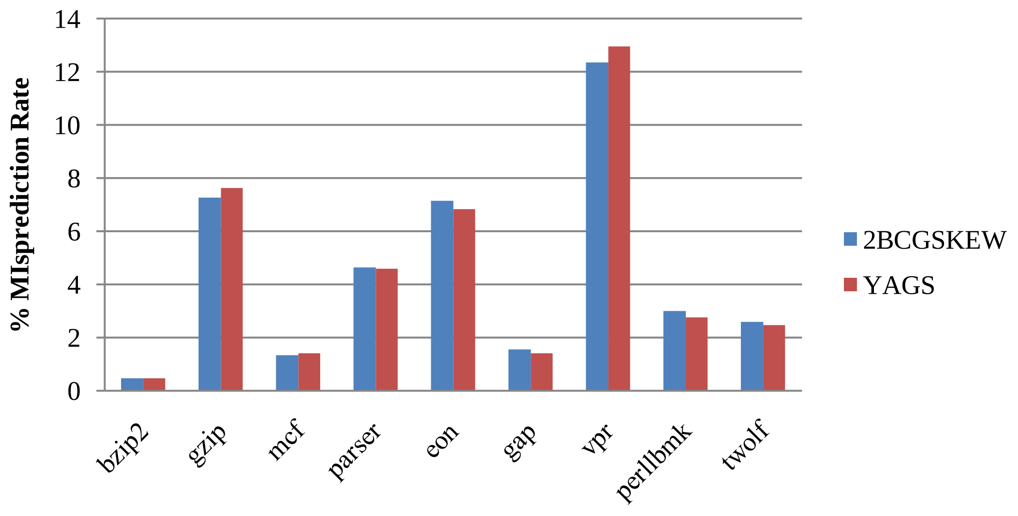

Figure 5.2.2.1 Misprediction rates with RIP initialization

Both the schemes show improvement from the worst case scenario in many benchmarks. While the maximum improvement on mispredicted branches provided by baseline scheme was 4.15% over its worst case configuration(for perlbmk), the maximum improvement achieved by YAGS was comparable, with absolutely less overall misprediction rates across all the benchmarks. It is worth noting that while the baseline scheme didn’t show any improvement for three out of the nine benchmarks simulated and worsened the misprediction rates for two and showed improvement for four, YAGS showed a worse performance for only mcf when compared to its worst case configuration without the history initialization, (by a minimal 0.7%) and showed improvement for all other benchmarks resulting in lower absolute accuracies except for mcf, gzip and vpr.

5.2.3 Broadcast scheme for DOE

The first step in the implementation of broadcast mechanism in DOE architecture was to broadcast the updates of the history based components of the baseline scheme. This means the updates of the gshare and gselect components of the baseline scheme are broadcast as and when the local updates happen. This is called the gshare broadcast. The underlying assumption of a gshare broadcast is that a branch which depends on a specific pattern of history repeats its behavior at any point in the program and hence could benefit from future speculative updates. A comparison of the baseline scheme with and without broadcast is shown in Figure 5.2.3.1.

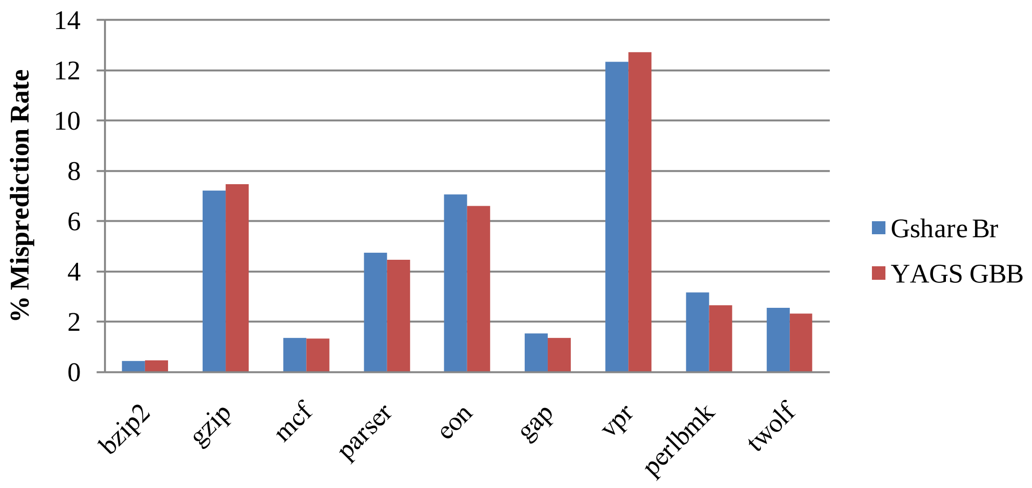

Figure 5.2.3.1 Gshare broadcast vs. 2BCgskew

The above comparison for only the baseline scheme shows the potential benefits of the broadcast scheme for gshare predictors. It should be noted that the gshare broadcast works on a core-to-core update basis and not on a core-to-buffer update basis. This model, when looked from an implementation point of view would suffer from the latencies that the updates take from reaching from one core to another. Also, the branches whose dependency is strictly on the history do not show a strong biased behavior with respect to the direction that they take. So buffering the updates over a period of time (which boosts the bias eventually) and sampling them at times might not help achieve the goal of initial perfect predictions as a thread runs. These considerations prompted the thesis to take a different direction and look at schemes where the inter-core latency should not affect immediate reads for prediction.

5.2.3.1 Final results for the Global Broadcast Buffer (GBB) scheme

Figures 5.2.3.1.1 and 5.2.3.1.2 show the misprediction rate comparisons of YAGS global broadcast buffer scheme (GBB) with the baseline scheme. The comparison is done for different size budgets. The results indicate that the broadcast and local update-read scheme using a GBB improves the misprediction rate from the best scheme evaluated so far. A look at the sizes of the tables for corresponding mispredictions reveals the effectiveness of the GBB scheme.

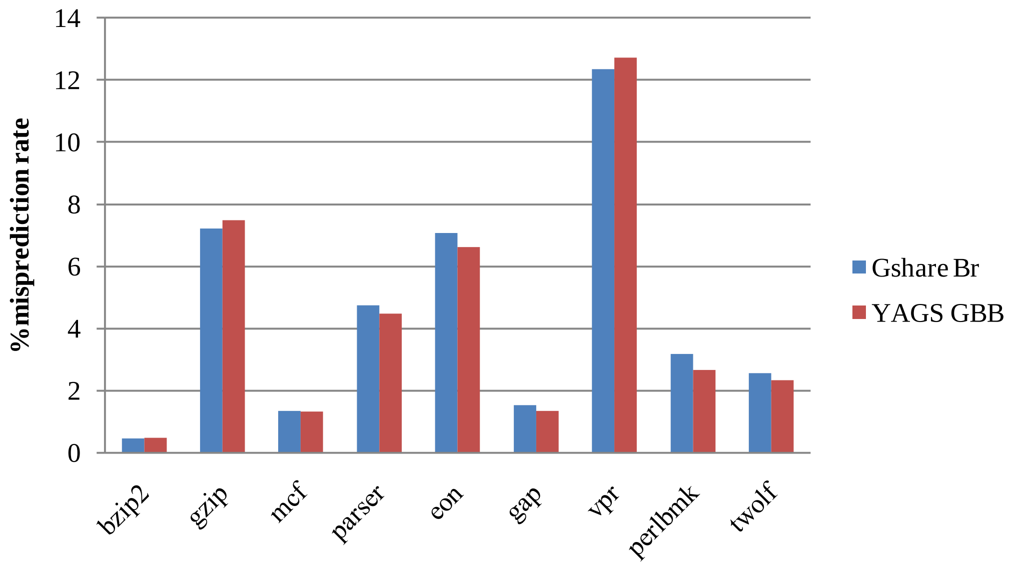

Figure 5.2.3.1.1 Misprediction rate comparisons for 1M baseline and 640KB GBB

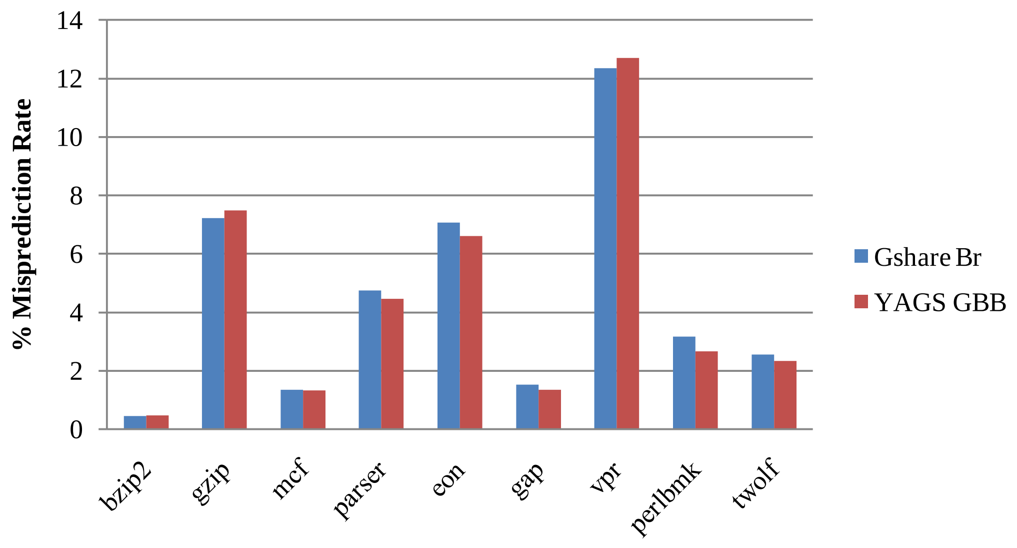

Figure 5.2.3.1.2 Misprediction rate comparisons for 512KB baseline and 640KB GBB

From the above figures it is very clear that the GBB scheme outperforms the 2BCgskew scheme with RIP initialization in under two thirds the size of the former for 7 out of the 9 benchmarks simulated. Figure 5.2.3.1.2 shows the comparison of misprediction rates for a configuration with 8 cores. The figure shows that the GBB scheme performs better than the 2BCgskew scheme with the GBB size remaining constant for both the 4 core and 8 core configurations. While the extent of scaling of the 2BCgskew predictor size is directly proportional to the increase in the number of cores, the size of the GBB stays the same irrespective of the number of cores. . Table 5.2.3.1.1 shows the improvement in the number of mispredicted branches of GBB scheme over the 2BCgskew scheme. The benchmarks which do not show improvement, gzip and vpr, and highly loop intensive and will benefit more from a Pag predictor than a Gshare.

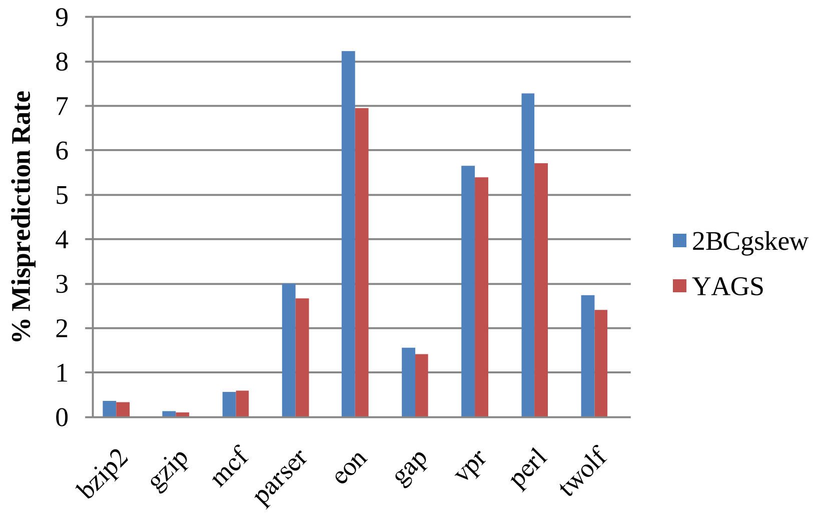

Figure 5.2.3.1.2 Misprediction Rate comparisons for 8 cores

	% Improvement in mispredicted branches
bzip2	1.44%
gzip	-2.43%
mcf	6.90%
parser	4.62%
eon	11.52%
gap	11.65%
vpr	-0.80%
perlbmk	16.95%
twolf	12.57%

Table 5.2.3.1.1 Improvement of misprediction rates of GBB over 2BCgskew

Finally we compare the very first broadcast scheme described here, gshare broadcast, and GBB scheme. The gshare broadcast is a core-to-core transfer mechanism and the sampling of the broadcast values is done all throughout the runtime as against thread boundaries for GBB.

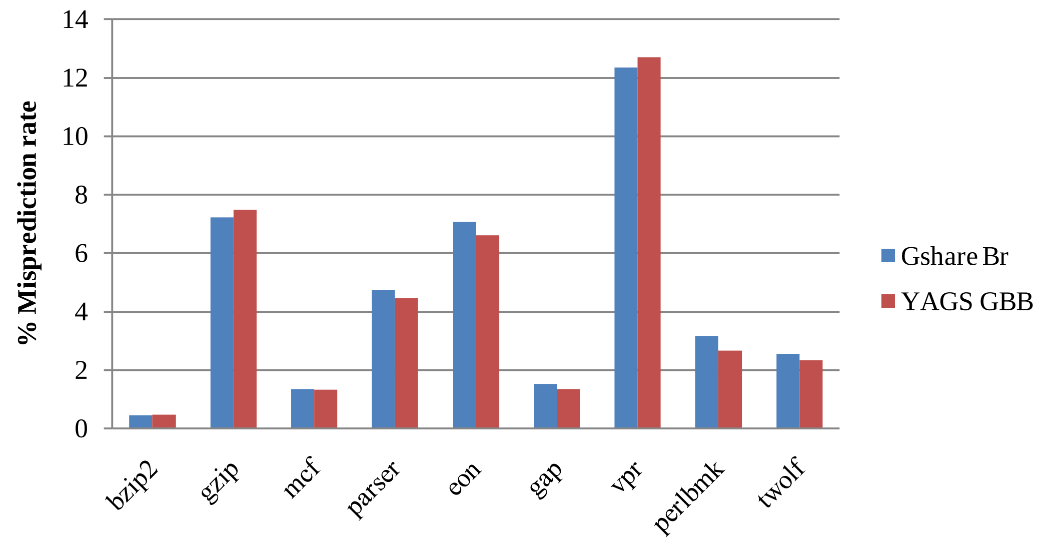

Figure 5.2.3.1.3 Comparison of gshare broadcast and GBB

Figure 5.2.3.1.3 shows that GBB outperforms gshare broadcast with less than 2/3rd the predictor size for all the benchmarks except vpr and gzip. Also it should be kept in mind that the gshare broadcast suffers from latency issues when modeled rightly. The study has not taken into account the effect of inter-core latency for gshare broadcast mechanism.

The results state that GBB scheme is a clear winner when existing prediction schemes are compared. One highlight of the GBB scheme is the fact that it is highly scalable. As the number of cores increase, the global buffer doesn’t need to scale with it and also the size of the local bimodal predictors in each core can be kept small. This is because of the following reasons.

1. Since the global buffer essentially buffers the biased behavior of the branches in an application, the size really doesn’t need to go up with the core size for a single application.
2. The local bimodal predictor handles the local updates in each thread. As the number of cores increase and thread hopping also goes up, these predictors essentially could be smaller since the initial read from the GBB gives bigger percentage of right predictions as the thread starts.
3. The latency of transfer to the global buffer is masked by the fact that the values from the global buffer are read only when a branch executes for the first time in a thread. This infrequent read of the global buffers results in a high probability of reading the reinforced bias value over time more than reading a value in the wrong bias due to inconsistent updates.

5.3 Indirect branches and Poisoned Branches

The thesis also led to the discovery of the fact that indirect branches significantly contribute to mispredictions in DOE. Even though conditional branch prediction accuracy is improved, to get performance corresponding to the improvement achieved, indirect branches also should be properly predicted. Indirect branches form a considerable chunk of the poisoned branches which are re-issued from the DTB (c.f. section 3.2.1) and can result in thread squashing rather than just pipeline flushes if mispredicted. Since the focus of the thesis is primarily on conditional branches, these areas are noted as observations. The GBB scheme shows improvement for poisoned conditional mispredictions by 25% for twolf, a benchmark which has considerable number of poisoned branches. The only other benchmark with significant number of poisoned branches is perlbmk where the improvement is close to 1%. All the other benchmarks have negligible poisoned mispredictions.

5.4 Hardware aspects and latency considerations

The GBB buffers speculative broadcast updates from all the cores at any point of time. This necessitates a multi-ported table for the implementation of the GBB. The number of read ports of this table will correspond to the number of cores used. For the write ports, there are two options. Either the number of ports equal to the number of cores could be used or a number less than the number of cores also can achieve comparable performance. This is due to the fact that the speculative broadcast updates from each core to the GBB carry out the purpose of boosting the biased nature of each branch. So the variation in the counter values is not going be a large deviation from either taken or not taken in the absence of a few writes from each core. The same principle holds good for the latency of the GBB updates from each core. The current thesis did not model the delays associated with the broadcasts from the cores to the GBB. This could be done effectively as part of an FPGA implementation in the future. For SpMT, the number of cores does not scale to a big number because the available thread level parallelism in single thread applications does not support a big number of parallel cores. For the GBB scheme where the number of read/write ports need to scale with the number of cores, this is ideal since the port number also does not need to scale beyond a limit.

Chapter 6

Conclusion and future work

6.1 Conclusion

The thesis proposes a new and simple scheme which makes use of the inherent biased nature of the branches in a program. A new scheme is proposed where a global table is used to buffer the updates from multiple cores speculatively to take advantage of the biased nature of branches. Different from the existing schemes, the broadcast scheme doesn't just look at how to resolve the issues relating to global history to improve prediction accuracy. It looks at a bigger domain outside the threads that can bring positive results. The broadcast scheme is scalable when compared to many other schemes examined, since it is not specifically correlated to threads in its table size or working principle. For 80% of the benchmarks simulated, the broadcast scheme provides the best prediction accuracies in a SpMT model.

6.2 Future work

The broadcast scheme could be applied to out-of-order cores and systems where the branch mispredictions are more crucial. To improve the accuracy of the dependent or poisoned branches, value prediction could be employed across threads. A mapping between the values predicted and the branch outcomes for dependent branches could potentially lead to a big improvement in dependent branch execution which would boost the SpMT performance for DOE. The broadcast scheme itself could be studied in more depth in terms of the way in which the GBB counter values are selectively updated. This could further improve the prediction accuracy.

References

[1] J. E. Smith, G. S. Sohi, ‘‘The Microarchitecture of Superscalar Processors,” *Proceedings of the IEEE,* vol. 83, no. 12, pp. 1,609-1,624, Dec. 1995.

[2] S. McFarling, ‘‘Combining branch predictors. Technical Report TN-36m,’’ Digital Western Research Laboratory, Jun. 1993.

[3] H. Akkary, M.A. Driscoll, ‘‘A dynamic multithreading processor,” *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture,* pp. 226-236, Nov. 1998.

[4] B. Choi, L. Porter, D. M. Tullsen, “Accurate branch prediction for short threads,” in *13th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 125–134, Mar. 2008.

[5] S. Hily, A. Seznec, ‘‘Branch Prediction and Simultaneous Multithreading,” *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pp. 6-13, Oct. 1996.

[6] C. Lee, I.K. Chen, T.N. Mudge, ‘‘The bi-mode branch predictor,” *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture,* pp. 4-13, Dec. 1997.

[7] A. N. Eden, T. Mudge, ‘‘The YAGS branch prediction scheme,” *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 69-77, Nov. 1998.

[8] J.E. Smith, ‘‘A study of branch prediction strategies,” *Proceedings of the 8th annual symposium on Computer Architecture*, pp. 125-140, May 1981.

[9] P. Marcuello, A. González, J. Tubella, ‘‘Speculative Multithreaded Processors,” *International Conference on Supercomputing’98*, pp. 77-84, July 1998.

[10] J. Gummaraju, M. Franklin. ‘‘Branch prediction in multi-threaded processors,” in *9th International Conference on Parallel Architectures and Compilation Techniques*, pp. 179, Oct. 2000.

[11] A. Seznec, P. Michaud, ‘‘De-aliased hybrid branch predictors,” *Technical Report RR-3618, Inria*, Feb. 1999.

[12] R. E. Kessler, ‘‘The alpha 21264 microprocessor,” *IEEE MICRO*, vol. 19, no. 2, pp. 24-36, Mar. 1999.

[13] A. Seznec, S. Felix, V. Krishnan, Y. Sazeides, ‘‘Design tradeoffs for the Alpha EV8 conditional branch predictor,” in *29th Annual International Symposium on Computer Architecture,* pp. 295-306, Jun. 2002.

[14] S. Nekkalapu, H. Akkary, K. Jothi, R. Retnamma, X. Song, ‘‘A simple latency tolerant processor,” *International Conference on Computer Design,* Oct. 2008.

[15] H. Akkary, K. Jothi, R. Retnamma, S. Nekkalapu, D. Hall, S. Shahidzadeh, ‘‘On the potential of latency tolerant execution in speculative multithreading,” *International Forum on Next-Generation Multicore/Manycore Technologies,* vol. 356, Nov. 2008.

[16] x86 Cycle Accurate Processor Simulation Design Infrastructure. http://www.ptlsim.org/

[17] T. Y. Yeh, Y. N. Patt, ‘‘Alternative implementations of two-level adaptive branch prediction,” in *19th International Symposium on Computer Architecture*, pp. 124-134, May 1992.

[18] G. Sohi, A. Roth, ‘‘Speculative Multithreaded Processors,” University of Wisconsin Madison White Paper, pp. 22-40.

[19] C. Iwama, N. Demus, B.S Sakai, H. Tanaka, ‘‘Improving Conditional Branch Prediction on Speculative Multithreading Architectures,” *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing,* pp. 413-417, Aug. 2001.

[20] T. Yeh, Y.N. Patt, ‘‘A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History,” *Proceedings of the 20th Annual International Symposium on Computer Architecture,* pp. 257-266, Jun. 1993.

[21] P.-Y. Chang, M. Evers, Y. Patt, ‘‘Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference**,”** *Proceedings of the 1996*

*Conference on Parallel Architectures and Compilation Techniques***,** pp. 48, Oct. 1996.

[22] P. Michaud, A. Seznec, R. Uhlig, ‘‘Trading Conflict and Capacity Aliasing in Conditional Branch Predictors,” *Proceedings of the 24th annual international symposium on Computer architecture*, pp. 292-303, May 1997.

[23] G. S. Sohi, S.E. Breach, T.N. Vijaykumar, ‘‘Multiscalar Processors,” *Proceedings of the 22nd annual international symposium on Computer architecture*, pp. 414-245, Jun. 1995.

[24] J.L. Hennessy, D.A. Patterson, ‘‘Computer Architecture: A Quantitative Approach, 3rd Edition”.

[25] D.M. Tullsen, S.J. Eggers, H.M. Levy, ‘‘Simultaneous multithreading: maximizing on-chip parallelism,” *Proceedings of the 22nd annual international symposium on Computer architecture*, pp. 392-403, Jun. 1995.

[26] M. Franklin, ‘‘The multiscalar Architecture : PhD Dissertation,” University of Wisconsin Madison, 1993.

[27] M.D. Hill, M.R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, pp. 33-38, July 2008.

[28] C. G. Qui.nones, C. Madriles, J. S´anchez, P. Marcuello, A. Gonz´alez, D.M. Tullsen, ‘‘Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices,” *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation***,** pp. 269-279, Jun. 2005.