Exploring Efficient SMT Branch Predictor Design

Matt Ramsay, Chris Feucht & Mikko H. Lipasti
ramsay@ece.wisc.edu, feuchtc@cae.wisc.edu, mikko@engr.wisc.edu
Department of Electrical & Computer Engineering
University of Wisconsin-Madison

Abstract: Branch prediction in simultaneous multithreaded processors is difficult because multiple independent threads interfere with shared prediction resources. We evaluate the prediction accuracy of four branch predictor configurations: 1) a totally shared predictor, 2) a completely split predictor, 3) a predictor with a shared history and split BHT, and 4) a predictor with a shared BHT and separate history registers, each for two static prediction schemes, a generic 2-bit predictor, a Gshare predictor, and a YAGS predictor. We simulate each combination listed with four threads executing a different benchmark and with each thread executing the same code.

We conclude that for an execution with unique threads, separating the branch history register for each thread allows for higher prediction accuracy by eliminating the negative aliasing between threads. Surprisingly, this splitting of predictor resources still performs better than a shared predictor even when each thread is running the same code. This study demonstrates that allotting each thread its own history register to access a shared predictor table gives performance close to that of a totally split predictor while using significantly less resources. Overall system performance, as measured in CPI, is only marginally affected by branch prediction accuracy in a multithreaded environment because thread-level parallelism allows for the hiding of long latency hazards, such as branch mispredicts. As such, we feel that branch prediction accuracy is not of peak importance when running multithreaded applications and that valuable development time and chip resources should be spent on other issues.

1. Introduction & Motivation

Simultaneous multithreading (SMT) is a form of multithreading [7] that seamlessly interleaves the processing of instructions from multiple threads of execution in a single, shared processor pipeline [15]. In a typical SMT system, a single set of branch predictors is shared between several concurrently running threads. Prior work has shown that this can lead to negative interference within the branch predictor as threads with very different control behavior interfere with shared prediction resources (e.g. [2,6,9]). By allocating an independent branch predictor for each thread, we demonstrate that there is potential to reduce negative aliasing, thus improving branch prediction accuracy. In contrast, the interference that occurs due to the sharing of branch predictors between threads could in fact be positive aliasing, where the leading thread trains the predictor for the trailing threads with similar control flow. If each thread were given its own branch predictor, the potential benefits of that positive aliasing would be lost and branch prediction accuracy would be reduced. Such positive interference may be demonstrated by allowing each thread to run the exact same program.

In this study, we determine the frequency of branch predictor interference for a set of multiprogrammed SMT workloads, whether the aliasing that occurs is positive or negative, and draw some conclusions as to an advisable design approach for branch predictors in SMT systems. This is accomplished by analyzing various branch predictor types as well as different combinations of branch predictor resource sharing. Each of these various configurations is then tested with workloads that are consistent with the cases presented above.

We also consider the tradeoffs between total processor performance and branch predictor size and complexity in an SMT system. Since there are other threads that can fill the void of a branch mispredict penalty, we hypothesize that less accurate, simple branch prediction schemes could be implemented in order to save resources with only a minimal performance loss. Prior work (e.g. [2,6,9]) has quantified the effect of sharing a branch predictor between threads on branch misprediction rate. Here, we also report performance effects, and find them to be surprisingly unimportant. Hence, we conclude that complex branch predictors are not called for

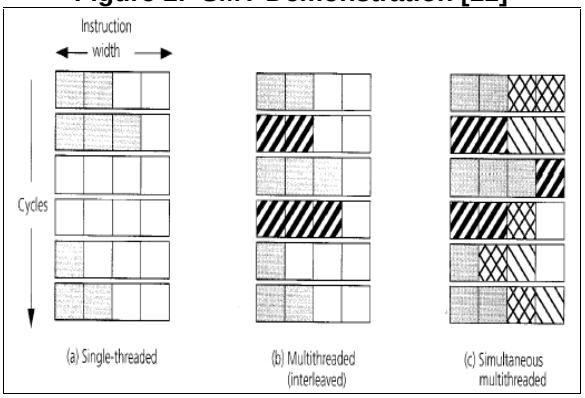
in SMT processors, since nearly the same performance (measured in CPI) can be achieved with a much simpler branch predictor, whereas the simpler design may enable faster cycle time or reduced design and verification time.

To describe these issues in further detail, the remainder of the paper is broken up as follows. Section 2 provides some basic information on simultaneous multithreading as well as additional motivation for this study. Section 3 explains the types of branch predictors that we studied. The overall test methodology that we employed for this work is covered in Section 4. Following that is a discussion of the test results in Section 5. Finally, Section 6 contains the conclusions that can be drawn from this study.

2. SMT Overview

In general, multithreading permits additional parallelism in a single processor by sharing the functional units across various threads of execution. In addition to the sharing of functional units, each thread must have a copy of the architected state. To combat this overhead, many other processor resources end up being shared by multiple threads. Accomplishing this sharing of resources also requires that the processor be able to perform switching between the threads efficiently, so as to not drastically affect performance. In fact, one can hide many of the memory stalls and control changes by switching threads at those occurrences.

Figure 1: SMT Demonstration [11]



The form of multithreading studied in our work is simultaneous multithreading (SMT) [10,15]. SMT processors are able to use resources that are provided for extracting instruction-level parallelism (e.g. rename registers, out-of-order issue) to also extract thread-level parallelism. An SMT processor uses multiple-issue and dynamic scheduling resources to schedule instructions from different threads that can operate in parallel. Since the threads execute independently, one can issue instructions from these various threads without considering dependencies between them.

Figure 1 shows a comparison of executions from a single-threaded processor to that of an SMT system. The middle picture shows an example of simple multithreading, which rotates through the available threads without exploiting the inter-thread parallelism. Overall, this solution does have some advantages over the single-threaded machine, but there are still many open functional units. This coarse-grained multithreading does not have the capability to allow different threads to work in parallel, and therefore many functional units are still underutilized. Thus, the SMT machine has the greatest potential for full resource utilization.

3. Branch Prediction Overview

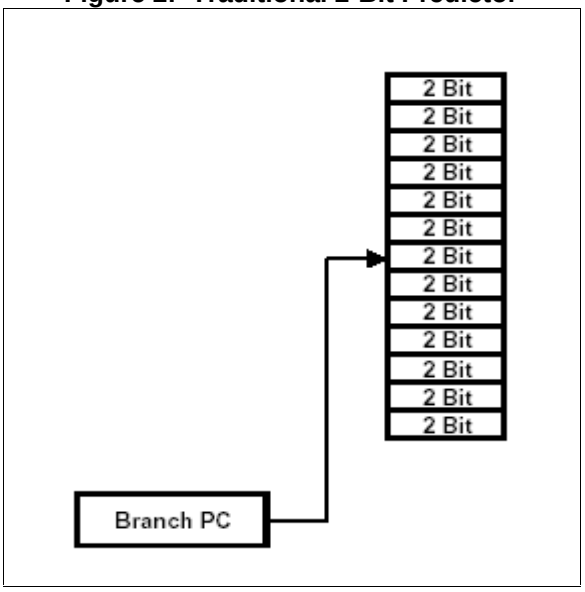
To show the validity of our work, we chose to apply our changes to five well-known branch prediction schemes. We study two static schemes, always taken and forward-not-taken-backward-taken [12]. We also examine three dynamic prediction schemes, the traditional 2-Bit predictor [12], the Gshare predictor [8], and the YAGS (Yet Another Global Scheme) predictor [5]. This section provides a brief overview of each prediction scheme.

3.1: Static Branch Predictors: The first static branch prediction scheme that we evaluated was the always taken scheme. Due to the high frequency of taken branches in most codes due to loop termination branches, this seemingly naive scheme has some validity. Building on the premise that most taken branches occur at the end of loops, a logical progression would be to predict that all backward branches are going to be taken and all forward branches not taken. In most codes, this scheme performs better than the always taken scheme. Although these static schemes have poor prediction accuracy, their major advantages are that they can be accessed instantly and they require no state to be maintained [12].

3.2: Traditional 2-Bit Predictor: One of the simplest dynamic branch prediction schemes is the traditional 2-bit predictor (Fig. 2) [12]. In this scheme, the lower bits of the branch instruction address index into a memory of 2-bit predictors called the branch-prediction buffer or branch

history table (BHT). Each BHT entry contains a saturating 2-bit counter that indicates whether the branch was recently taken or not. The prediction is read from the BHT in order to speculatively determine whether to begin fetching instructions from the taken or not-taken path. If the 2-bit counter is greater than 0x01, then the branch is predicted taken; else it is predicted not taken. Later in the pipeline when the branch outcome is determined, the predictor is updated by incrementing the counter if the branch was taken or decrementing it if the branch was not taken. Identical 2-bit saturating counters are used for branch prediction in the Gshare [8] and YAGS schemes [5].

Figure 2: Traditional 2-Bit Predictor



The 2-bit scheme's simplicity and speed make it an attractive option. However, its simple indexing method does not take advantage of global history information. Correlating branch prediction on the recent branch history has proven to increase prediction accuracy in most programs. [16]

3.3: Gshare Branch Predictor: The Gshare branch prediction scheme (Fig. 3) uses a recent global branch outcome history as well as the branch instruction address to index the BHT [8]. Indexing the BHT with the XOR of the branch history and address eliminates a significant amount of the aliasing that occurs using the traditional 2-bit prediction scheme and also takes advantage of recent branch history information.

Figure 3: Gshare Predictor

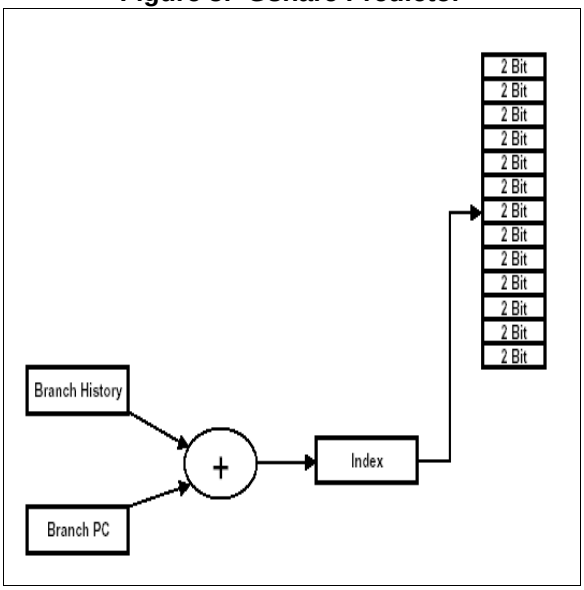
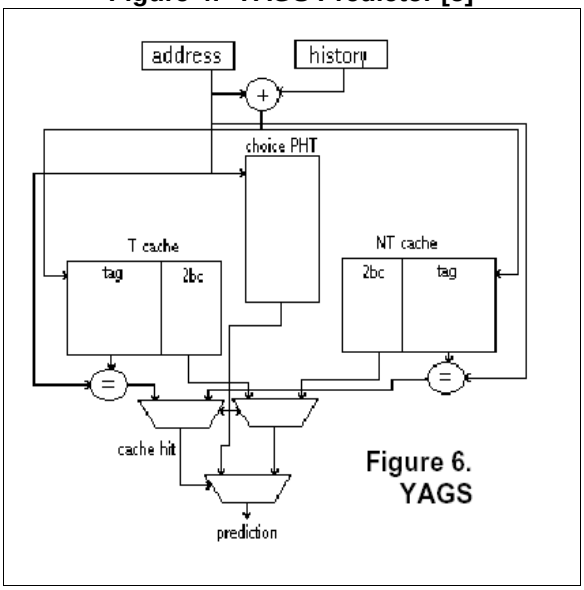


Figure 4: YAGS Predictor [5]



3.4: YAGS Branch Predictor: As can be seen in Figure 4, the YAGS predictor [5] is a much more complicated and expensive prediction scheme that contains three different predictors:

- BHT for biased taken branches
- BHT for biased not taken branches
- Choice PHT: BHT to choose between the taken and not taken predictors

In this scheme, the branch address is used to index into the Choice PHT and for the tag comparison from the bias taken and not taken BHTs. The taken and not taken BHTs are indexed strictly by the XOR of the branch history

Table 1: Tested Machine Parameters

# of Threads	4	# Instructions Simulated	~40M
# Address Spaces	4	L1 Latency	1 cycle
# Bits in Branch History	12	L2 Latency	10 cycles
# of BT Entries	4096	Memory Latency	200 cycles
# bits in Indirect History	10	L1 Size	32KB
# IT Entries	1024	L1 Assoc.	DM
Pipeline Depth	15	L1 Block Size	64B
Machine Width	4	L2 Size	1MB
Max Issue Window	64	L2 Assoc.	4
# Physical Registers	512	L2 Block Size	128B

and branch address. The biased taken and not taken BHTs both produce their prediction for the branch outcome and one is chosen by the prediction produced by the PHT. If there is a hit on the address in the chosen predictor, the entry's counter is used as the branch prediction. If there is a miss, the way prediction from the PHT is used as the overall prediction. Using this combination of the branch history and address eliminates virtually all aliasing in traditional single-threaded programs. This scheme also performs well for multi-threaded applications because there is less history interference between threads (Graphs 5 & 8). As usual, this higher prediction accuracy does not come for free. The YAGS prediction scheme requires an amazing amount of state, as there are three predictors, two of which hold large address tags in addition to 2-bit counters. It is reasonable to assume that the access to this predictor is much slower and more power intensive than the other prediction schemes.

3.5: Indirect Branch Prediction: An indirect branch, also known as a Jump-Register instruction, is an unconditional branch that receives its target address from a register. In many cases the value of the target register is not known when the indirect branch issues. Therefore, performance can be gained from accurately predicting the target of the indirect branch and sequentially fetching instructions from the predicted target rather than stalling fetch while the target is computed [3]. In general, indirect branches are much more difficult to predict than conditional branches because the predictor must select the correct target address rather than just a taken/not taken result. Aliasing in the indirect branch predictor compounds the problem. Aliasing is much more of a problem in the indirect branch predictor because the BTB, called the ITB (Indirect Target Buffer) in the indirect predictor, must hold full addresses rather than just 2-bit saturating counters and therefore must have much fewer entries. The indirect branch predictor that we study is indexed in the same manner as the Gshare branch predictor, with the XOR of the branch history and the branch instruction address. This hashing helps with the aliasing problem, and for the codes we simulated indirect branch prediction is very accurate. This result is not typical. [3]

4. Test Methodology

4.1: Simulation Environment: For our study we used the SIM-MULTI simulator, developed in part by Craig Zilles at UW–Madison, which is based on the Simplescalar distribution [1]. It supports multiprogramming of several independent programs, so it is capable of being passed multiple input programs and assigning those to various threads. We used optimized Spec2000 [14] Alpha binaries that were provided with the simulator [13].

For our experiments we configured the simulator to run four threads. An important feature of the simulator with regard to branch prediction is that when a mispredict occurs, only the instructions associated with the thread that was mispredicted are squashed. Key machine parameters used in our simulations are shown in Table 1.

4.2: Simulated Benchmark Description: To effectively test with the above simulator, a series of benchmarks would be needed. Given that the multiprogramming approach was used in the design of the simulator, SPEC2000 benchmarks would be used for the evaluation purposes. These were used unmodified with the standard reference input files for testing and simulated for the first 40M instructions.

To explore performance under various computational environments, two program

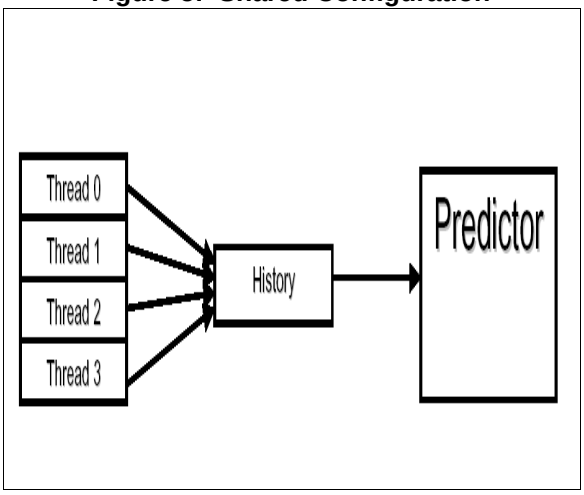
models are tested. One configuration that can exist in an SMT is when each thread is running a different program. This configuration simulates a typical multitasking environment. Another configuration is where the processor is running multiple copies of the same program such that each thread has a separate copy of the program. This setup simulates a typical web server.

Given the overall set of SPEC benchmarks, we decided that for the first case we would run two integer and two floating-point benchmarks (ammp, crafty, equake, gcc). This would result in four unique threads and overall a good mix of performance. To simulate the second case, one of the selected integer benchmarks (crafty) would be used and copied for each of the threads. The integer benchmark was chosen because it was felt that this would better mimic the type of workload that a web server would employ.

4.3: Branch Predictor Configurations: In order to observe the amount of branch prediction interference that occurs between threads in an SMT, we simulated eight different configurations of conditional and indirect branch predictors. The various predictor configurations are described below and were used for each of the predictor types. In each diagram, the branch predictor is shown as two blocks: the "History" block which encompasses the recent branch history register and the "Predictor" block which encompasses the rest of the predictor. Although each of the diagrams shows a generic predictor that includes a history register, the same configuration principles can be applied to the traditional 2-bit predictor even though no history register is used. There obviously can be only one configuration for the static prediction schemes so we compare that single case to each configuration of the dynamic schemes when appropriate.

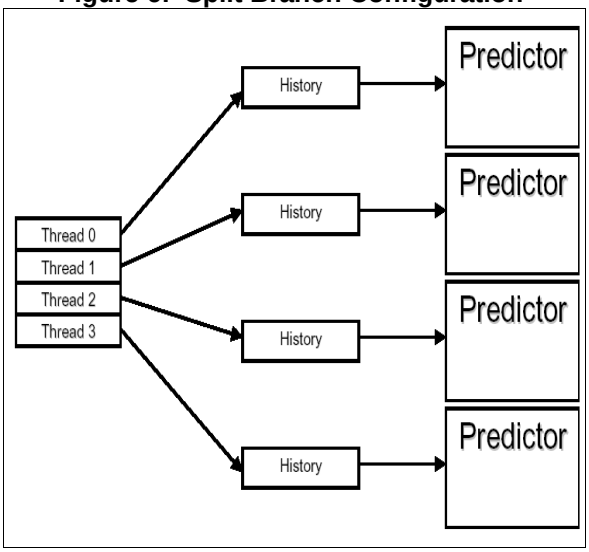
4.3.1: Shared Configuration: The most resource conservative branch predictor configuration for an SMT is a totally shared predictor (Fig. 5). In this case, each thread shares both the history register and BHT. This configuration allows for the most interference between threads. This interference can occur both in the history register and in the BHT. As you would expect, the configuration that allows for the most interference requires the least state.

Figure 5: Shared Configuration



4.3.2: Split Branch Configuration: The next logical configuration to test was providing each thread with its own predictor (Fig. 6). This configuration completely eliminates interference between threads. In this case, the predictor acts exactly as it would in a single-threaded environment. Again, not surprisingly, the configuration that eliminates all interference requires the most state, especially with the duplication of the YAGS predictor.

Figure 6: Split Branch Configuration

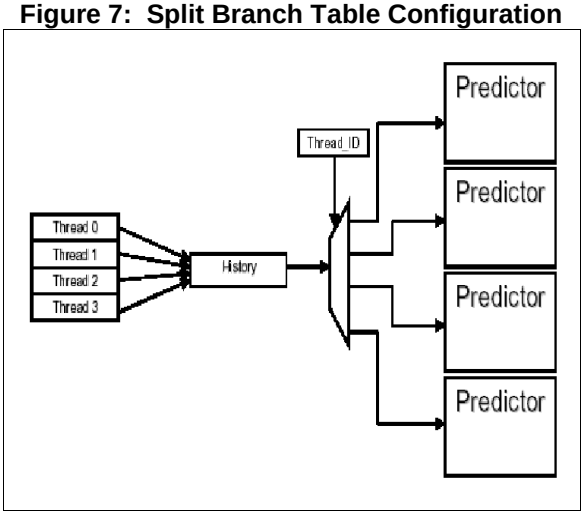


Note this study is principally concerned with the interference that occurs in the branch predictor due to shared history between threads and not reduction in branch prediction accuracy due to effectively reduced capacity of the predictor when multiple threads are competing for space. Therefore, when simulating split

predictors, we model configurations of the same size as the shared predictor to eliminate second order effects from reduced capacity in the split predictors.

4.3.3: Split Branch Table Configuration: The third configuration that we simulated does a partial split of the branch predictor (Fig. 7). In this case, each thread accesses a common branch history register, but then indexes into its own predictor. This configuration allows interference only in the branch history register.

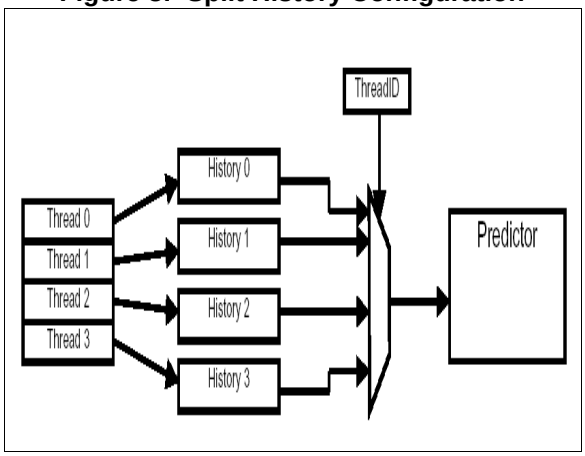
Figure 7: Split Branch Table Configuration



4.3.4: Split History Configuration: The final branch predictor configuration simulated does the opposite partial split of predictor resources (Fig. 8). This configuration allots each thread its own history register while indexing into a common predictor. This configuration again only allows interference at one of the two possible places, in the predictor. By only replicating the branch history register, a small resource, instead of the predictor, a much larger resource, the split history configuration eliminates one of the sources of interference in a much more cost and space efficient manner than the split branch table configuration. The hope of this configuration is that interference in the larger predictor will have less effect than interference in the smaller branch history register.

4.3.5: Indirect Predictor Configuration: To this point, we have established test cases that vary the configuration of the branch predictor. We simulate each of these four configurations with a unified and split indirect branch predictor, giving us eight test configurations. It is important to note that replicating the indirect branch predictor table is expensive because it contains full addresses rather than 2-bit saturating counters. Any performance benefit gained by splitting the indirect predictor should be weighed against that fact.

Figure 8: Split History Configuration



5. Results

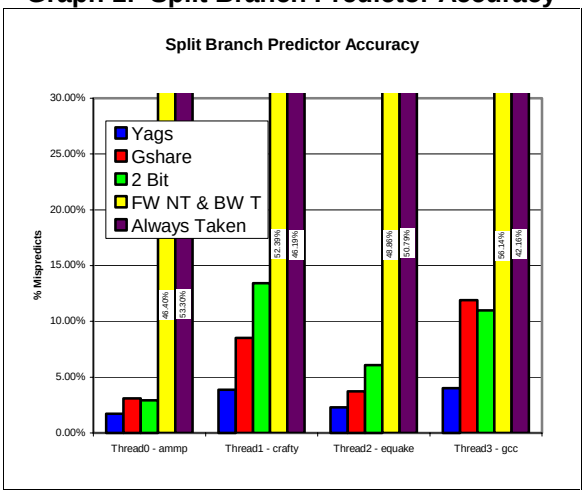
This section presents the results of our experiments along with a commentary discussing their significance. The graphs containing data from experiments using the YAGS prediction scheme that are presented in sections 5.2 – 5.4 have eight bars for each thread. They represent the following cases from left to right:

- Shared Configuration with Unified Indirect Branch Predictor
- Split Branch Configuration with Unified Indirect Branch Predictor
- Split Branch Table Configuration with Unified Indirect Branch Predictor
- Split History Configuration with Unified Indirect Branch Predictor
- Shared Configuration with Split Indirect Branch Predictor
- Split Branch Configuration with Split Indirect Branch Predictor
- Split Branch Table Configuration with Split Indirect Branch Predictor
- Split History Configuration with Split Indirect Branch Predictor

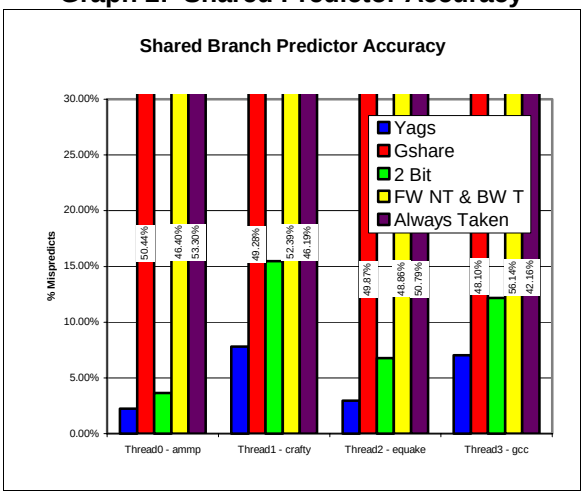
Graphs describing the Gshare prediction scheme only contain four bars per thread, which correspond to the first four bullets above. Since the Split Branch Table and Split History predictor configurations are not applicable for the 2-bit scheme that does not use a history register, only the first two cases are shown in these graphs.

5.1: Branch Predictor Scheme Prediction Accuracy: In order to evaluate our proposed configurations of the five targeted branch prediction schemes, we must first establish their baseline prediction accuracy. Graph 1 shows the misprediction rate of the five branch prediction schemes when run in isolation, the split branch configuration (Fig. 6). This configuration shows how each branch predictor would perform in a single-threaded environment. This data matches intuition, as the YAGS predictor performs the best, followed by the Gshare and finally the traditional 2-bit predictor. The static schemes perform much worse than each of the dynamic schemes, which is also expected.

Graph 1: Split Branch Predictor Accuracy



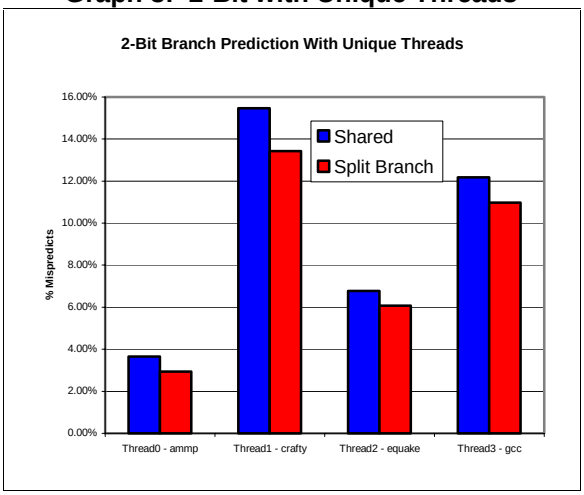
Graph 2: Shared Predictor Accuracy



Graph 2 shows how these same branch prediction schemes perform in a shared configuration (Fig. 5), where the total branch predictor is shared between threads. This data demonstrates the interference that occurs when branch prediction resources are shared between unrelated threads. This interference reduces the accuracy of each scheme. The YAGS and 2-bit prediction schemes are similarly affected by sharing, but the Gshare scheme is affected significantly by thread interference, even to the point of being outperformed by some of the static schemes. The Gshare prediction scheme is affected much more because interference happens both at the branch history register and in the BHT. The 2-bit predictor only incurs interference in the BHT, therefore its performance holds, as does that of the YAGS scheme where the complexity of the scheme overwhelms the interference. The accuracy for the static schemes does not change because their predictions are not affected by program behavior.

5.2: Branch Prediction with Unique Threads: This section shows the prediction accuracy for each dynamic prediction scheme running in each of the four predictor configurations with the threads running a unique program, as in a multi-tasking environment. Graph 3 shows the prediction accuracy of the traditional 2-bit predictor for each configuration mentioned earlier. Providing each thread with its own predictor increases prediction accuracy for each of the benchmarks by eliminating interference between the unrelated threads.

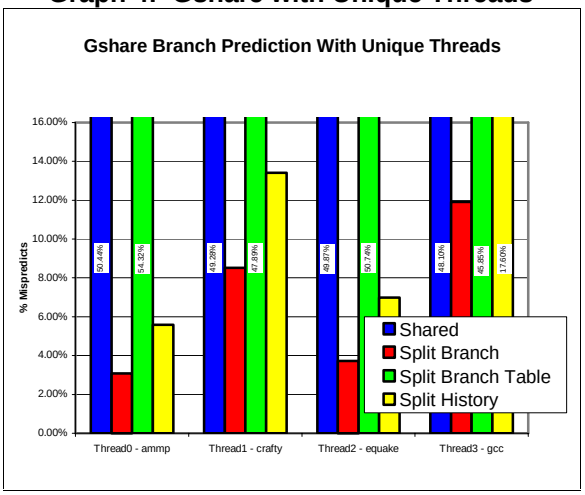
Graph 3: 2-Bit with Unique Threads



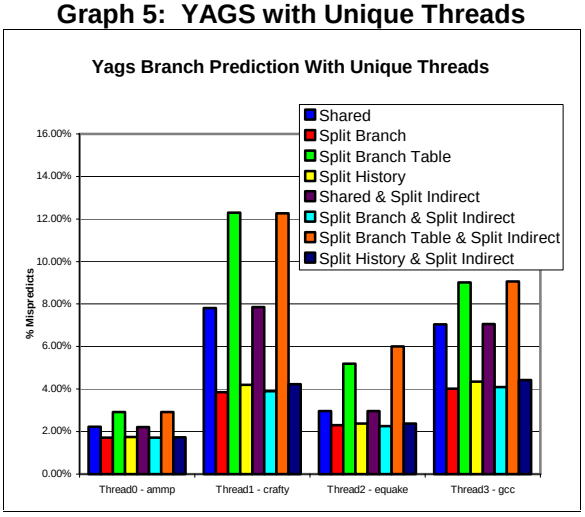
Graph 4 presents each configuration for the Gshare predictor. For similar reasons to the 2-bit scheme, completely dividing the predictor increases prediction accuracy by a significant

amount. However, when the branch history register is shared between threads (shared and split branch table configurations), accuracy suffers greatly, producing near 50% mispredicts in all cases. This is due to the significant effect of interference in a small, shared resource. As expected, the highest prediction accuracy for the Gshare predictor is seen when the predictor is completely split, thus eliminating the aliasing

Graph 4: Gshare with Unique Threads



Graph 5: YAGS with Unique Threads



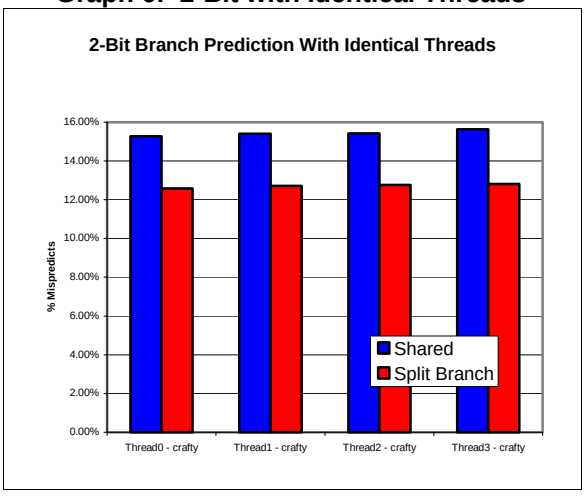
that can occur in two places in the predictor. In addition, the split history configuration is only slightly less accurate than the split branch configuration. In this case, the accuracy is maintained without having to duplicate the BHT, a significant expense.

Sharing only the branch history register has a similar negative effect on the YAGS predictor (Graph 5). However, in this case, much less accuracy is gained by a complete split of the branch predictor. We believe that this is due to the complexity of the YAGS scheme, which is able to eliminate most aliasing even when multiple threads access a shared predictor. Similar to the Gshare scheme, the split history configuration again retains most of the accuracy provided by the split branch configuration while using significantly less resources.

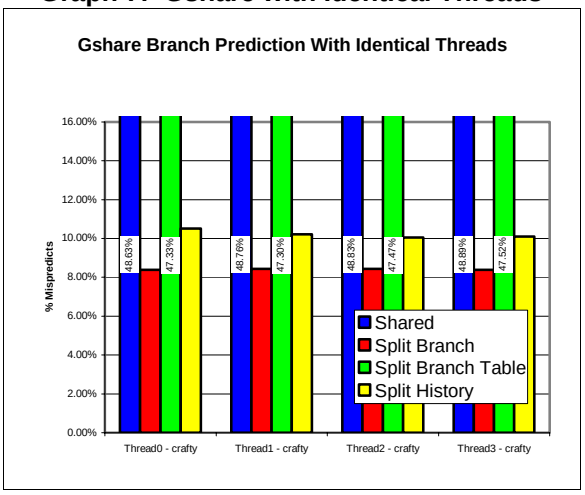
Another interesting point is that the splitting of the indirect branch predictor has virtually no effect on the branch prediction accuracy, even though it shares the branch history register with the branch predictor (Graph 5). This is due to the fact that indirect branches are infrequent in our simulations. It is also worth noting that indirect branch prediction accuracy was stable across all simulation configurations.

5.3: Branch Prediction with Identical Threads: This section shows results from experiments using identical threads. This corresponds to a web server application. One might expect that in this case a shared predictor would benefit from positive aliasing between the identical threads. But in fact, for each branch prediction scheme, the split predictor is still much more accurate than the shared configuration (Graphs 6, 7, 8). In each case, the shared and split branch table configurations still perform very poorly. These results again show that the split history configuration provides similar prediction accuracy to the split branch configuration, even more closely mirroring it than in the unique thread studies. This occurs because in many cases the thread that leads the execution will train the BHT for the trailing threads. When these trailing threads encounter the branch, their

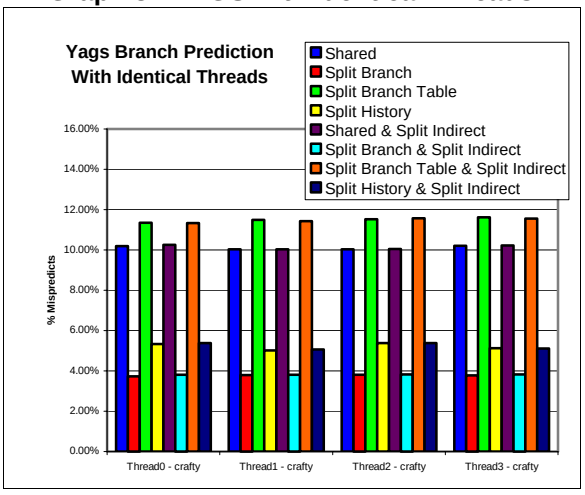
Graph 6: 2-Bit with Identical Threads



Graph 7: Gshare with Identical Threads



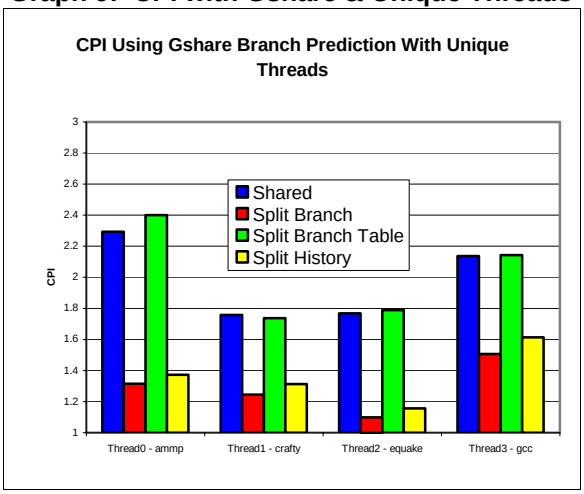
Graph 8: YAGS with Identical Threads



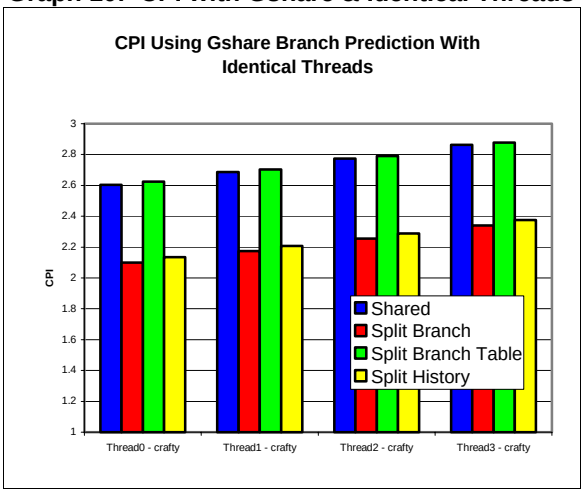
individual history registers will likely be the same as the leading thread at that branch because of similar control flow between identical threads and thus allow the access of the same 2-bit predictor in the BHT. These trailing threads will then benefit from the BHT training of the leading thread. This data also suggests that branch prediction design should be uniform across a broad range of workloads.

5.4: Prediction Effects on Performance (CPI): The data in this section shows that overall performance is not significantly affected by branch prediction accuracy in an SMT. This is in contrast to traditional superscalar machines where branch mispredicts cause complete pipeline flushes and cause noticeable slowdowns. This point is demonstrated by the fact that in a single-threaded environment the crafty benchmark shows a 6% increase in CPI using the Gshare scheme and an 11% CPI increase using the 2-bit scheme as compared to the YAGS scheme. In an SMT environment, these CPI increases are 2% and 5% respectively with the same branch prediction accuracy for each scheme (split branch configuration). As mentioned earlier, this occurs because other threads can fill the void when one thread encounters a mispredict penalty. From this data, we draw the conclusion that branch prediction accuracy is much less important in an SMT processor than in a traditional superscalar machine, when the focus is on multithreaded application performance.

Graph 9: CPI with Gshare & Unique Threads



Graph 10: CPI with Gshare & Identical Threads

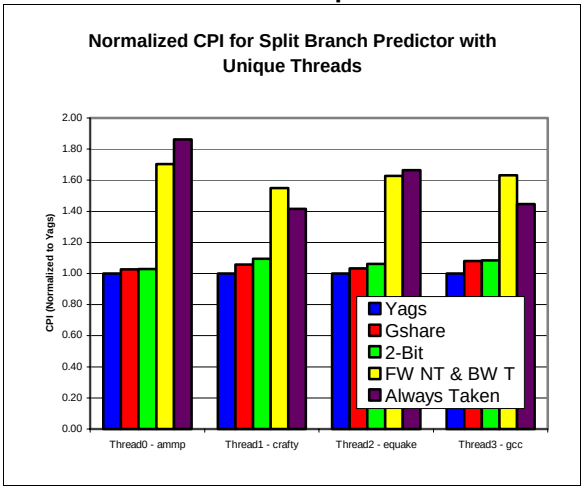


As shown by Graphs 9 and 10, the per-thread CPI remains relatively constant across the split history and split branch predictor configurations. Although only the Gshare scheme is shown, this is the case across all

branch prediction schemes. The previous prediction accuracy graphs in conjunction with these two graphs, indicate that sharing only the branch history register between threads is detrimental to performance. Given the fact this configuration performs poorly across all prediction schemes, it will not be discussed further.

When running identical threads, we find that performance degrades dramatically (Graph 9 vs. Graph 10). This is due to the fact that our multiprogrammed approach for SMT does not allow the threads to share any state in the cache. Hence, the cache miss rates experienced by the processors double due to conflict and capacity misses between the identical threads. The unique threads are better able to share the available cache memory and have a lower overall cache miss rate, which correlates with the higher observed throughput. This fact is magnified when using the crafty benchmark for the identical thread studies because the crafty benchmark causes more cache misses than the other three benchmarks tested. Running four threads of crafty makes the overall cache miss rate even higher, resulting in the poor performance seen in Graph 10. Additionally**,** there have been studies that

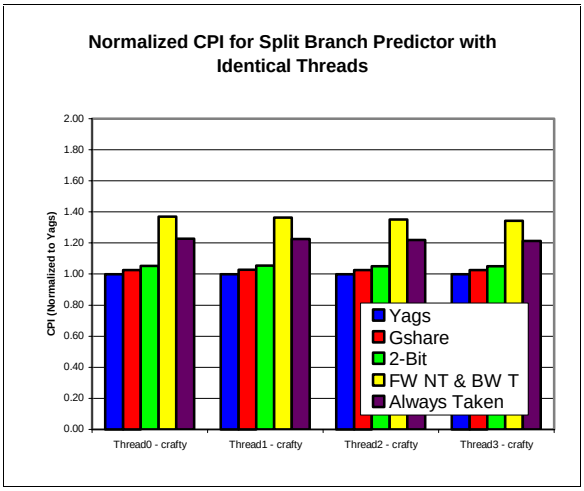
Graph 11: Normalized CPI for Split Branch Predictor with Unique Threads



prove that some benchmarks do not perform well when run together in an SMT [4]. We hypothesize that crafty falls into that category.

The next set of graphs indicates the relative change in performance as the branch predictor is simplified. The performance of each prediction scheme is normalized to that of the

Graph 12: Normalized CPI for Split Branch Predictor with Identical Threads



YAGS scheme on a thread-by-thread basis. This data is presented for both the split branch (Graphs 11 & 12) and split history (Graphs 13 & 14) predictor configurations and simulated for both the unique and identical thread cases. This data indicates that the low accuracy of static prediction schemes cannot be tolerated in the presence of other independently executing threads in SMTs. While these schemes provide substantial savings in chip real estate, power consumption, and access time, these benefits cannot compensate for lack of delivered performance.

The other cases, however, yield some interesting results. Analyzing Graphs 11 and 12, one can see that best-case slowdown for the Gshare scheme using the split branch configuration is in the unique thread case with approximately a 2% increase in CPI. The impact appears to be similar with the 2-bit scheme with its best-case CPI increase residing at 3%. The performance delta between these schemes and YAGS is only slightly more significant with the split history configuration, as seen in Graphs 13 and 14, where the slowdown is 7% and 4%, respectively.

These results are important to note because one now has some difficult choices to make. By simplifying the predictor from YAGS down to one of the other schemes, significant savings can be realized in several areas. As mentioned earlier, the YAGS predictor has three tables that need to be referenced, compared to the other schemes that simply have a single table. This additional hardware can be rather power hungry. Another major concern is that this more advanced scheme will take longer to access, and thus

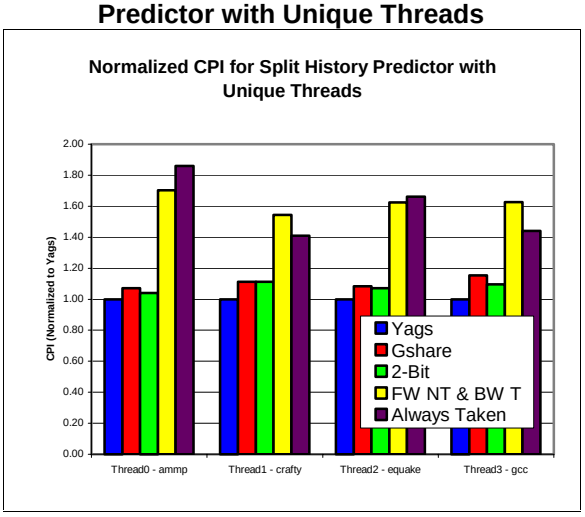
Table 2: Instructions Executed Per Thread (YAGS)

Thread ID - Benchmark	Instructions Committed			
	Shared	Split Branch	Split Branch Table	Split History
Thread0 - ammp	40466600	40396520	42201142	40432869
Thread1 - crafty	42779617	43952717	43025427	43877029
Thread2 - equake	48617686	48617686	48617686	48617686
Thread3 - gcc	36218492	37101290	36246587	37070529

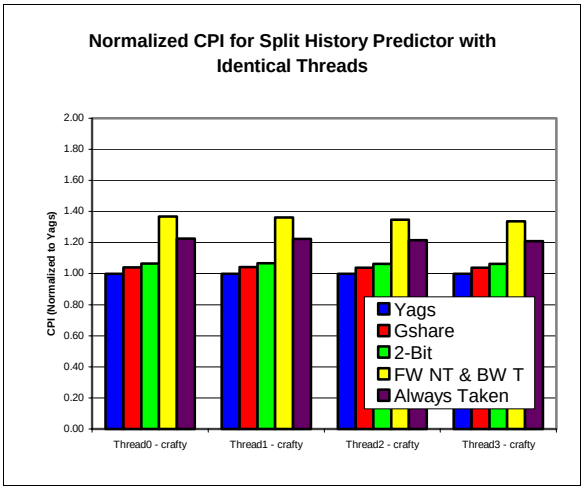
there may be some additional speed-ups for the simpler schemes that are not considered by our simulator because of additional branch prediction delay in the front end. Of course, these gains must all be balanced by the loss of performance that was observed above.

Given this information, we propose that one should utilize either the Gshare or the 2-bit predictor over implementing a more complicated scheme. With the relative size of these schemes, as compared to YAGS, one could even possibly implement one of these in a split branch predictor configuration in the same or less amount space as a shared YAGS predictor, if the number of threads is small. This split would help to improve the performance of either of these simplified schemes, as evident from Graphs 3 and 4. Of course, this split branch configuration would still have less power dissipation than the unified YAGS, since it is likely that only one thread would need to access the predictor at a given time, and the other predictors could remain idle. It is also evident that the split history configuration retains most of the benefits of the split branch configuration with much less resource usage. Overall, the split history configuration that utilizes one of the simpler dynamic prediction schemes seems to be the best balance between processor performance, power dissipation, and chip area when the goal is to design a multithreaded processor capable of running concurrent threads efficiently.

Graph 13: Normalized CPI for Split History Predictor with Unique Threads



Graph 14: Normalized CPI for Split Predictor and Identical Threads



An important note about the CPI numbers presented here is that they are only close estimates and direct comparison between simulations is not completely valid. This is caused by the fact that multithreaded executions, like multi-processor simulations, are not deterministic when system parameters are changed. In our experiments, branch predictor accuracy is altered, causing branch mispredicts to be encountered at different times, thus affecting which thread executes during a particular cycle. This allows for the possibility of each thread committing a different number of instructions in multiple simulations of the same workload. This fact alone makes comparison of CPI results across simulations less dependable. This point is demonstrated by the data in Table 2. Since we are only trying to prove that CPI is relatively unchanged across simulations, we feel that our conclusions still hold.

6. Conclusions

As a result of our studies, we have drawn several conclusions. First, multithreaded executions can interfere with speculation, particularly branch prediction accuracy, because of aliasing in shared prediction resources. Second, a multithreaded simulation that executes unique codes on each thread is hindered by the negative aliasing that occurs by sharing a branch predictor. This aliasing can be combated by allowing each thread to access its own pieces of the branch prediction structure. In the case where each thread runs a copy of the same program, where intuition would suggest that positive aliasing between threads would help a shared predictor, the split predictor configuration still performs better. This suggests that the branch predictor of an SMT can be designed independent of the thread relationship. In our studies, we also looked at the effects of similar strategies on the prediction of indirect branches. We found that changes to the predictor structure had little effect on the accuracy of indirect branch prediction. We attribute this to the relative infrequency of indirect branches and their high predictability in our simulations.

More importantly, we conclude that branch prediction accuracy is less important in an SMT system than in a traditional superscalar processor. Misprediction ratios that increased by factors of two or three only caused slowdowns of as little as 2%. This conclusion is demonstrated by the fact that the normalized per-thread CPI is relatively constant, even for cases of very poor dynamic branch prediction accuracy. This phenomenon occurs because other threads fill in the processing void left when a long latency hazard is encountered by one thread. Finally, because of the relative unimportance of branch prediction accuracy in a multithreaded environment, we conclude that valuable development time and on-chip resources should be applied to other more important issues. Simple predictors, such as the Gshare and 2-bit schemes, appear to perform adequately in terms of CPI and can save significantly on chip real estate and power dissipation.

References:

1. Burger, D. and T. Austin. “The SimpleScalar Tool Set. Version 2.0.” Technical Report, University of Wisconsin-Madison Computer Science Department, 1997.
2. Cain, H., Rajwar, R., Marden, M., and Lipasti, M. “An Architectural Evaluation of Java TPC-W.” In *Proceedings of HPCA-7*, January 2001.
3. Driesen, K. and U. Holzle. “Accurate Indirect Branch Prediction.” *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, July 1998, pages 167-178.
4. F.N. Eskesen, et.al. “Performance Analysis of Simultaneous Multithreading in a PowerPC-based Processor.” *Workshop on Duplication, Destructing, and Debunking*, Anchorage, AK, May 2003.
5. Eden, A. N. and T. Mudge. “The YAGS Branch Prediction Scheme.” *In Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69-77, 1998.
6. Hily, Sebastien and Andre Seznec. “Branch Prediction and Simultaneous Multithreading.” *Proceedings of PACT ’96*, October 1996, pages 169-179.
7. Hennessy, John and David Patterson. *Computer Architecture A Quantitative Approach: Third Edition*. San Francisco: Morgan Kaufmann, 2003.
8. McFarling, S. “Combining Branch Predictors.” Technical Report WRL TN-36, 1993.
9. Seznec, A., Felix S., Krishnan, V., and Sazeides, Y. “Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor.” In *Proceedings of ISCA-29*, May 2002.
10. Simulation and Modeling of a Simultaneous Multithreading Processor, D. M. Tullsen, *In the 22nd Annual Computer Measurement Group Conference*, December, 1996.
11. Smith, B. J. “Architecture and Applications of the HEP Multiprocessor Computer System.” *Proceedings of the International Society for Optical Engineering*, 1981, pages 241-248.
12. Smith, J.E.. “A Study of branch prediction strategies.” In *Proceedings of ISCA-8*. 1981.
13. www.simplescalar.com
14. www.spec.org

15. Tullsen, D., Eggers, S.J, and Levy, H.M. “Simultaneous Multithreading: Maximizing On-Chip Parallelism.” In *Proceedings of ISCA-22*, 1995.
16. T. Y. Yeh and Y. N. Patt. “Two-level Adaptive Training Branch Prediction.” *Proceedings of the 24th Annual Workshop on Microprogramming (MICRO-24)*, Albuquerque, NM, p. 55-60, Dec. 1991.
17. Zilles, Craig B., Joel Emer, and Gurindar Sohi. “The Use of Multithreading for Exception Handling.” *Proceedings of Micro-32*, 1999, pages 219-229.