Evaluating Branch Predictors on an SMT Processor

David Mulvihill, Matthew Allen
CS 752 Project Report
University of Wisconsin – Madison

Abstract

Simultaneous multithreading (SMT) provides significant increases in microprocessor throughput by issuing instructions from multiple threads per clock cycle. SMT can be realized in a wide-issue superscalar with a modest increase in resources, because much of the hardware is shared among the multiple thread contexts. Branch prediction accuracy, a key component of microprocessor performance, can suffer greatly from destructive interference caused by multiple threads sharing the same branch prediction hardware. Although SMT processors are able to hide branch latencies by scheduling instructions from other threads, branch prediction accuracy is still important in SMT processors to reduce latencies of individual threads and to provide a diverse mix of threads to the instruction scheduler. This paper evaluates several well-known branch prediction schemes, and examines some modifications to address problems caused by sharing of branch prediction hardware in SMT.

1 Introduction

Modern high-performance superscalar processors attempt to maximize instruction-level parallelism (ILP) by issuing multiple instructions per clock cycle. But increasing ILP through wide-issue yields diminishing returns as control and data hazards limit the sustained throughput to much less than the peak. Simultaneous multithreading (SMT) attempts to increase superscalar processor throughput by executing instructions from multiple threads each cycle, thus exploiting thread-level parallelism (TLP) on-chip. This is accomplished by maintaining an architectural state per thread; instructions are issued from multiple threads per cycle, and the execution back-end is dynamically shared among the thread contexts.

SMT yields benefits over superscalar and chip-level multiprocessors (CMP). While the latter two focus on ILP and TLP, respectively, an SMT processor can dynamically adjust ILP and TLP as the situation demands. In multithreaded environments, a 4-way SMT processor can achieve more than 200% increase in throughput over an equivalent issue superscalar processor [1].

Although SMT has an architectural state per thread, many of the execution resources must be shared among threads. This includes the instruction queue, caches, TLB, branch predictor and execution units. Because the processor can sustain a significantly higher throughput, there is a greater potential for bottlenecks than in a non-SMT processor. [1] attacks the problem of efficient multithreaded instruction fetch by providing an even mix of instructions from available threads. It was also determined that register file size has a significant impact, as a large register file is needed to sustain active threads.

What is less certain is the impact of branch prediction on an SMT processor. It is well known that branch prediction is very important in a speculative superscalar processor; mis-speculations decrease performance by wasting cycles on unused, speculative instructions. SMT, on the other hand, is less sensitive to branch prediction, because instructions can still issue and execute from other threads if one thread mis-speculates. However, we believe that branch prediction is still important for high performance; not only do threads now share the branch prediction hardware, but inter-thread history and prediction table aliasing can impact prediction rate. Furthermore, Swanson et. al. have shown that speculation is necessary to provide the rich mix of threads required to achieve high throughput in an SMT [8]. An SMT processor is not aimed solely at multithreaded workloads; the 4-way SMT Alpha EV8 was still designed around single process performance, and thus could not ignore high-performance branch prediction [2]. Therefore, we believe that choices made in branch predictor configuration is still important in an SMT processor.

To understand the impact of branch prediction on performance, we evaluate several branch prediction techniques on a simulated SMT processor with four thread contexts. The gshare, gskew, two-level, and bimodal predictors are evaluated, and several variations of each are studied. We also explore prediction accuracy as a function of predictor size.

The remainder of the paper is organized as follows: Section 2 discusses the branch prediction schemes that were simulated. Section 3 discusses our simulation methodology. Section 4 discusses prediction accuracy for single threads. Section 5 examines branch prediction with four simultaneous threads. Section 6 describes related work, and Section 7 provides the conclusions of this paper.

2 Branch Prediction Strategies

Branch predictors use a combination of branch addresses, branch history, and state information to predict the outcome of branch instructions. Both local history (the history of the branch instruction being predicted) and global history (the history of all branch instructions) can be used to make predictions.

The simplest branch prediction scheme is the *bimodal predictor*, which uses part of the branch address to hash into a table of two-bit saturating counters [3]. When a branch is taken, the counter is incremented; when a branch is not taken the counter is decremented. The prediction is made using the sign bit of the counter.

The *two-level predictor* leverages local history to improve the accuracy of the bimodal scheme [4]. This strategy uses part of the branch address to index into a table of branch histories. The value in this table is used to index into a second table of two-bit counters, and the prediction is made in the same way as the bimodal scheme.

In many programs, branch outcomes are highly correlated to the outcomes of other recent branches. This observation led to the development of branch predictors that use global history to improve prediction rate. The *gshare* branch predictor uses a global history register to store the outcome of the recent branches in a program [5]. A table of two-bit counters is indexed using the exclusive OR of branch address bits and the global history register.

Another global scheme is the *gskew* predictor [6]. The gskew scheme attempts to alleviate aliasing (when multiple branches hash to the same location in a predictor table) by using three tables that are indexed using a skewing function. The prediction is made by the majority vote of the three predictors. The skewing functions are designed so that aliases usually occur in only one of the three tables. *The partial update policy* is used to prevent aliases from interfering with each other. On a misprediction, all three predictor tables are updated. On a correct prediction, only the predictors that correctly predicted the outcome are udpated. The incorrect predictor is not updated, assuming that it is aliased with another branch, and that changing its value could result in a misprediction of the alias.

3 Methodology

The CPPSIM simulator was used to conduct our evaluation of branch prediction on an SMT processor. CPPSIM is an SMT extension of SimpleScalar written by Craig Zilles. The simulated SMT machine supports four thread contexts, 8-way instruction issue, and has 64 KB L1 data cache, a 64 KB L1 instruction cache, and a 2 MB L2 cache. This configuration is representative of a high-end SMT processor that could be realized in the near future. In order to make a fair comparison, each branch predictor was configured with the nearest possible size to 64 Kbits.

To simulate a multiprogrammed workload, the programs comprising the SPEC benchmark suite were run simultaneously. Eight integer SPEC benchmarks were run: gap, gcc, gzip, parser, perlbmk, twolf, vortex, and vpr. The programs were run in two groups of four, and the results averaged. Four floating point SPEC benchmarks were evaluated: apsi, equake, galgel, and mesa. All programs were fast-fowarded for one billion instructions and simulated for an additional one billion instructions.

Multithreaded programs have different branch prediction characteristics than multiprogrammed workloads because branches in the independent threads often occur at the same addresses. Unfortunately, multithreaded benchmarks were not readily available for CPPSIM. In order to provide a comparison between multiprogrammed and multithreaded workloads, we attempted to simulate multithreaded programs by running four simultaneous instances of the same SPEC benchmark with different inputs. The three programs with at least four input files were used: gcc, gzip, and perlbmk; the results were averaged to produce the multithreaded result.

4 Baseline Single Thread Performance

The fundamental goal of SMT is to maximize throughput by leveraging thread-level parallelism to the extent possible without degrading performance when a single thread is running on the processor. Thus, any evaluation of branch prediction must consider the performance of programs running as the only thread on the processor. Performance of single threads also provides a baseline for comparison with measurements of multiple threads.

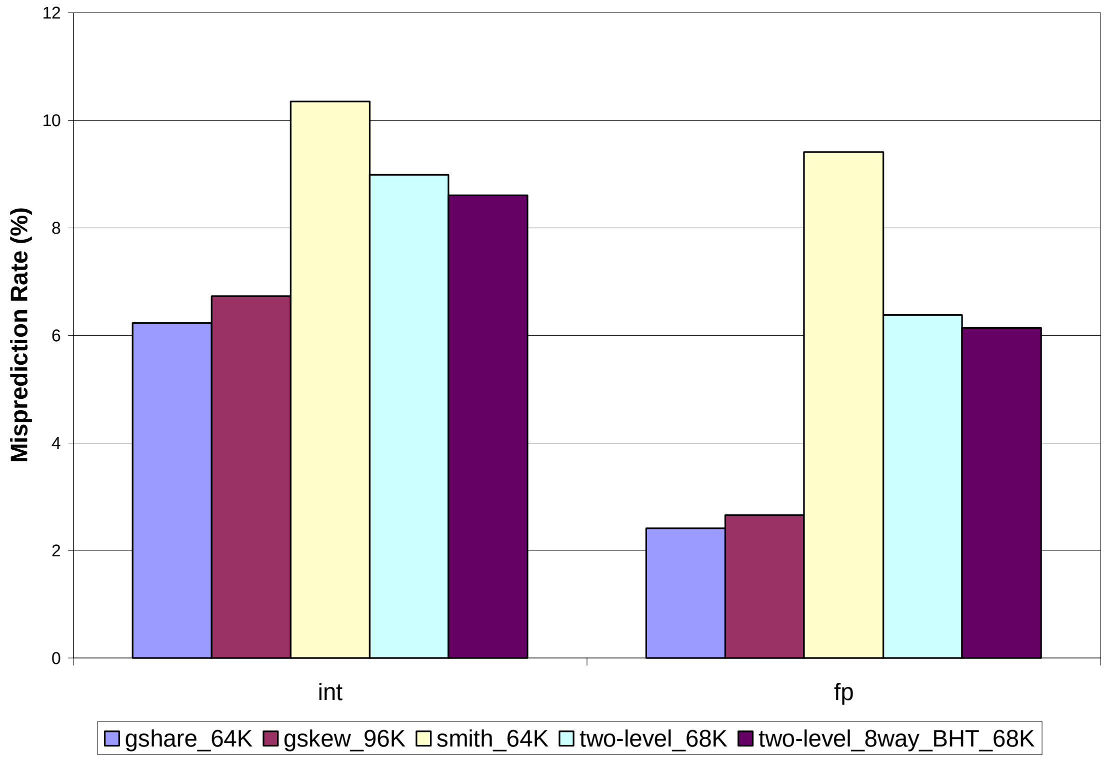

Figure 1: Single Thread Misprediction Rate

Figure 1 shows the misprediction rates of several branch prediction schemes for the integer and floating point benchmarks. The gshare and gskew predictors clearly illustrate the advantage of using global history. The two-level predictor is the next-best performer. Two varieties of the two-level predictor are shown: the predictor that uses the PC to hash into the BHT, and a predictor with an 8-way set associative BHT, which provides a very modest improvement. The bimodal predictor, which does not benefit from the use of history information, is the worst performer.

Figure 2 shows IPC for the same set of predictors. Clearly, IPC is a strong function of the branch predictor accuracy. This is the expected result; mispredictions cause large delays because cycles are wasted performing useless instructions, and the correct instruction must be fetched before computation can resume.

5 Performance With Simultaneous Threads

First we evaluate multithreaded performance using predictors unmodified for SMT, in which all threads compete for predictor resources. This may expose the pathological case in which inter-thread aliasing and the multiplexing of the predictor harms prediction accuracy and performance.

In Figure 3, the first result that is of interest is gshare with a shared global history register (gshare_shared_64K). Clearly, sharing the global history register among all threads breaks the performance of the predictor, since it can no longer correlate the behavior of global branches within a particular program. This results in misprediction rates exceeding 40%; likewise, IPC drops significantly, from 4.6 to 3.75 in the SPECint multiprogrammed test. Thus the first optimization that can be made for schemes that use global history is to maintain a global history register per thread. Since there are only four thread contexts, the extra hardware required to realize these additional registers should be minimal. The per-thread global history register optimization yields significantly improved performance. For gshare (gshare_64K), the misprediction is 8%, compared with 6% for the single thread SPECint workload, and floating point prediction rates are equal.

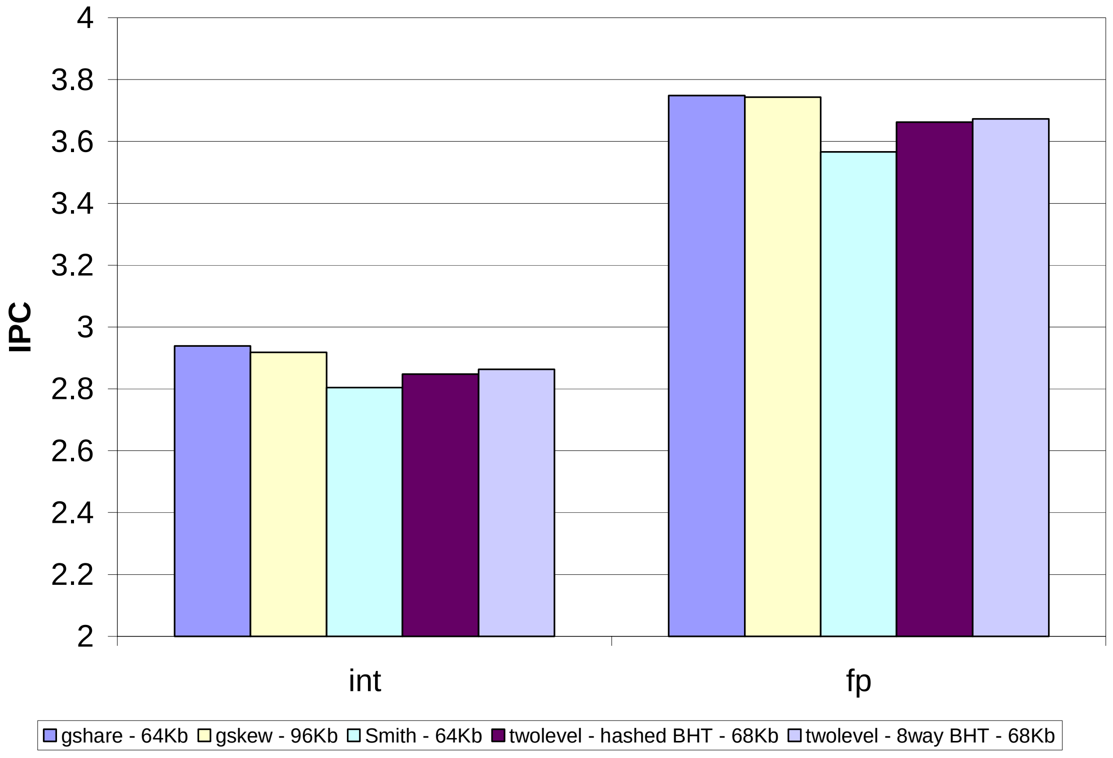

Figure 2: Single Thread IPC

Another modification to gshare that was evaluated was the static division of the pattern table per-thread to eliminate inter-thread pattern aliasing. This results in little improvement to the integer and floating-point multiprogrammed benchmarks. The multithreaded workload sees a slight decrease in prediction accuracy. This is likely due to the fact that a multithreaded workload may actually benefit from inter-thread aliasing, since the threads are running from the same address space. This, combined with the smaller amount of pattern table capacity available to each thread, results in worse performance for the multithreaded workload.

The gskew prediction scheme does not perform as well as the gshare predictor. Although gskew attempts to reduce aliasing with skewed associativity, each prediction is stored in three separate pattern tables, causing the pattern table capacity to be effectively one-third that of gshare. Even with a larger predictor size (96K vs. 64K for gshare), gskew still produces 1% higher misprediction rate for the integer benchmarks.

In an attempt to improve the performance of gskew, we added the two-bit thread id as an input to the skewing function (gskew_tid). This has the effect of reducing both the branch history and branch address used by one bit each. The effect of this modification was neutral; we believe the skewing function is already eliminating most of the aliasing, and the degraded performance relative to gshare is due to the reduction in pattern table size.

While the base two-level predictor (2level_68K) loses some accuracy for the integer benchmarks when compared to the single thread case, floating-point misprediction rate actually decreases from 6% to 5% in the multiprogrammed environment. This could be due constructive aliasing in the pattern table between threads due to floating-point loop-intensive behavior that likely produces similar history patterns and branch outcomes.

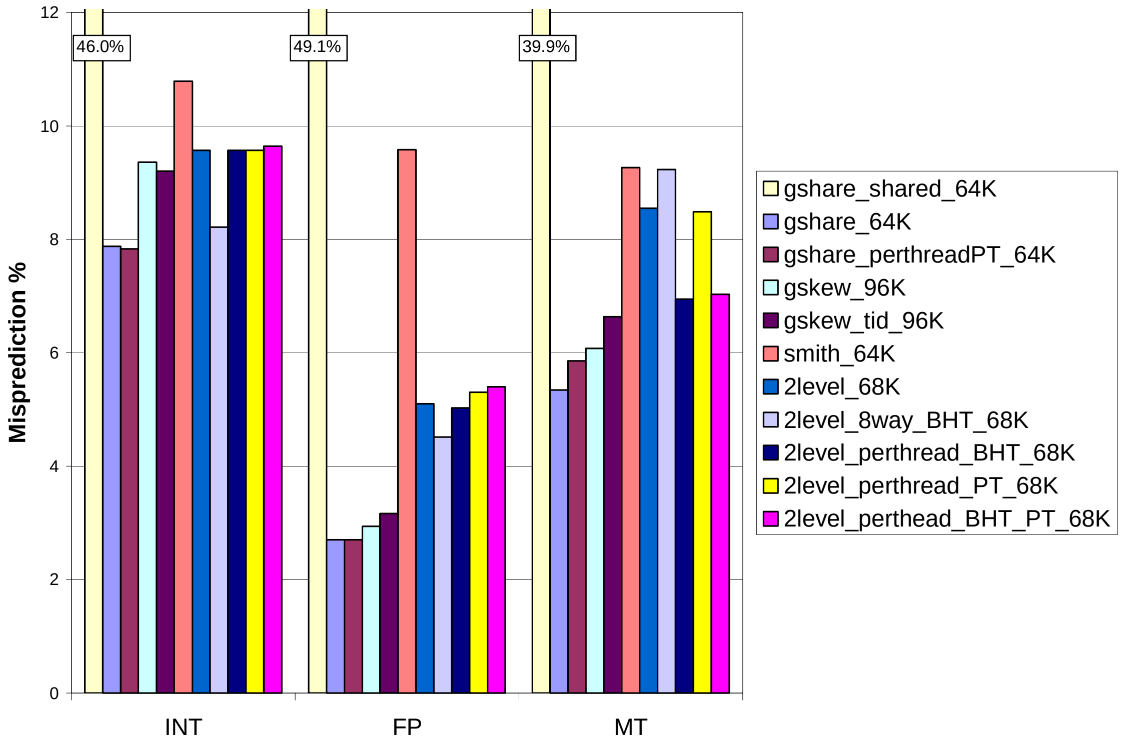

Figure 3: Misprediction rate with four simultaneous threads for the multiprogrammed (INT, FP) and multithreaded (MT) workloads.

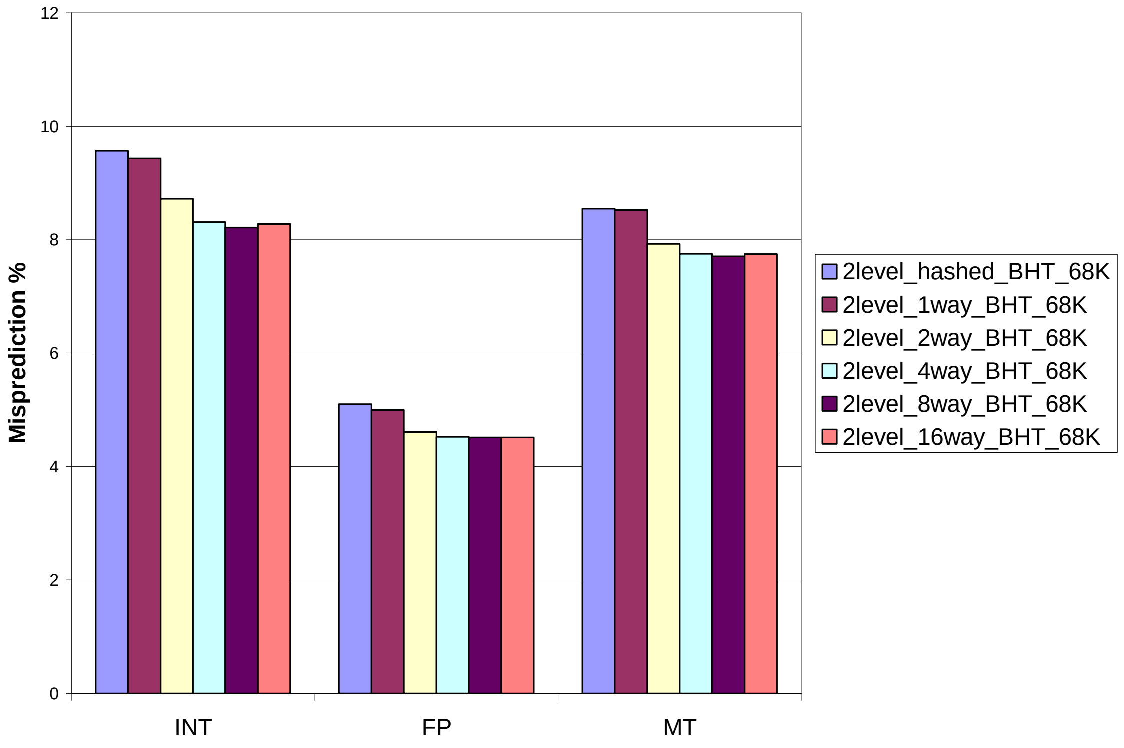

Figure 4: Misprediction rates vs. associativity of the BHT in a two-level branch predictor.

The optimizations made to the two-level predictor evaluated the effect of limiting or eliminating aliasing in the history and pattern tables. The base two-level predictor employs a hashing method to index into history table, so there is a chance that two branches from different threads may conflict in a history entry if the lower-order bits are the same. By utilizing a set-associative cache for the history table, two branches from different threads will conflict only if their branch addresses match. This approach does not improve performance for one thread, but does improve the multi-programmed case. Figure 4 shows misprediction rate as a function of associativity for the two-level pattern table. Prediction accuracy improves 0.5-1.5% with 4-way associativity; higher associativity does not significantly improve prediction. Note that the overhead of tag bits was not considered for this scheme, because a branch target buffer contains the same tag bits. If a branch target buffer is not present, the two-level predictor with a set-associative BHT would require more state bits than other predictors in the comparison.

We also evaluated per-thread branch history tables, pattern tables, and the combination of both for the two-level predictor. While dividing the pattern table does not improve prediction accuracy, dividing the branch history table yields a substantial benefit in the multithreaded workload. Because the threads are running from the same address space, the elimination of inter-thread aliasing results in improved prediction. This confirms the belief of Seznec et al. that interference for local history entries can pollute histories and patterns for threads spawned by a single application [2].

It should be noted that while the two-level predictor updates history entries upon a branch's commit, the gshare and gskew predictors perform speculative history updates. Skadron et al. describe a method to perform speculative history updates and the cleanup methods necessary when mispredictions occur [8]. It was found that speculative updates for global and local history improve IPC by 10% in SPECint95. Therefore, the lack of speculative history updates in our two-level predictor must be considered when comparing its performance to that of gshare and gskew. While implementing the checkpointing mechanism for gshare's global history is straightforward [8], doing the same with the two-level predictor's large history table is difficult, especially in light of the large number of branches an SMT processor has to resolve each cycle. Therefore, it is unlikely that speculative history updates for a two-level predictor would be possible with our machine configuration.

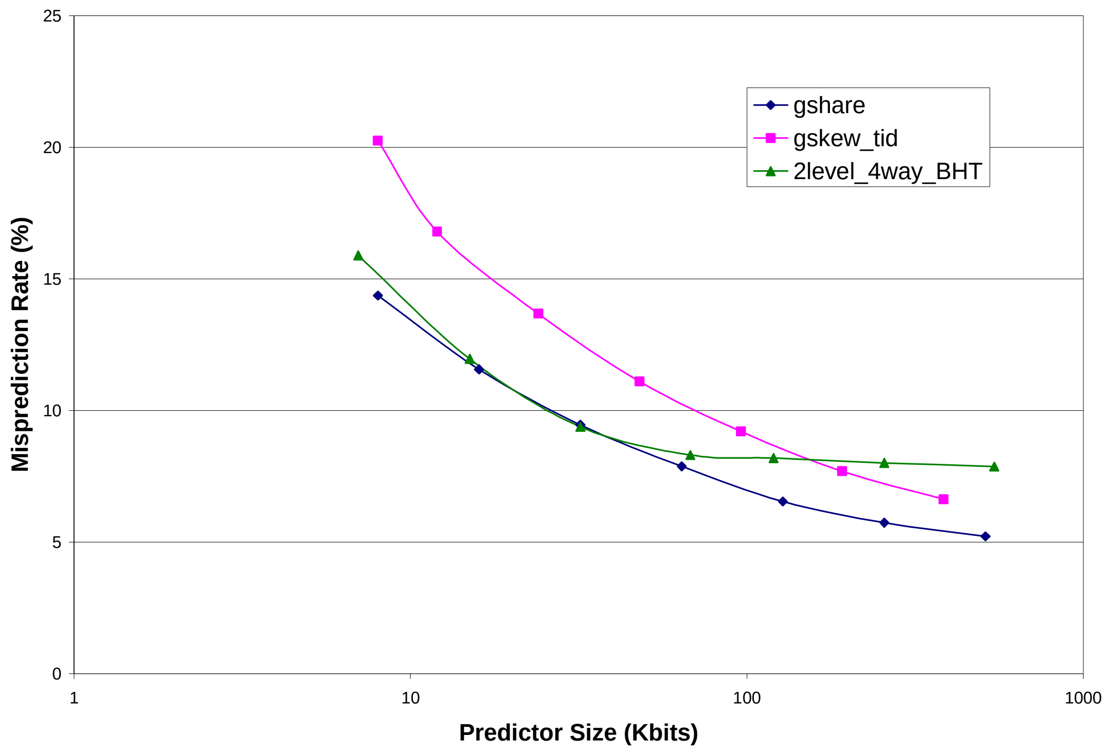

Figure 5: Prediction accuracy vs. predictor size for the gshare, gskew, and two-level branch (with set-associative BHT) predictors for the SPECint benchmarks with 4 simultaneous threads.

In addition to evaluating branch prediction accuracy as a function of prediction strategy, we also examined the effect of state bit budget on accuracy. Figure 5 shows the prediction accuracies of the gshare, gskew with thread id, and the two-level predictor with 4-way associative BHT for the integer benchmarks. The gshare and two-level predictors perform equivalently for smaller predictor sizes, but as the bit budget grows, gshare continues to improve while two-level flattens out. Gskew performs poorly for small predictor sizes due to the limited amount of pattern table capacity, but outperforms the two-level scheme for predictor sizes greater than 100Kbits.

Figure 6 shows prediction accuracy as a function of predictor size for floating point programs. The gshare predictor outperforms the others once again. However, for floating point applications gskew outperforms the two-level scheme even at small predictor sizes.

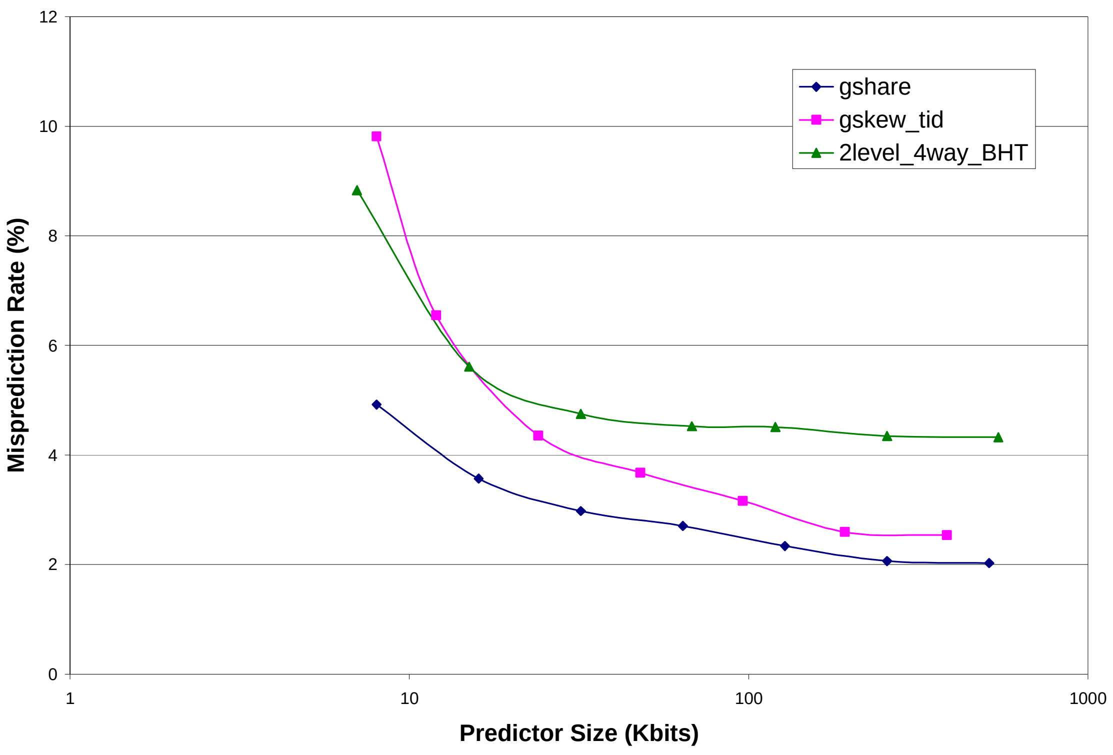

Figure 6: Prediction accuracy vs. predictor size for the gshare, gskew, and two-level branch (with set-associative BHT) predictors for the SPECfp benchmarks with 4 simultaneous threads.

Finally, we examine IPC for the simulated benchmarks. Figure 7 shows IPC for the same benchmarks shown in Figure 3. Despite the varying branch prediction rates, SMT is successful in hiding the branch latencies by scheduling instructions from other threads. Only the very poorly performing gshare predictor with the shared global history register degrades throughput.

6 Related Work

Tullsen et al. discuss the limitations of branch prediction to SMT performance in [1]. They find that perfect branch prediction improves performance by 25% at 1 thread, 15% at 4 threads, and 9% at 8 threads.

Seznec et al. provide a detailed discussion of the Alpha EV8 branch predictor in [2]. The EV8 design is an 8-wide out-of-order superscalar, featuring 4 SMT contexts. The EV8 uses a 2Bc-gskew branch predictor, which is a hybrid of the gskew and bimodal schemes. The paper emphasizes the importance of using global history in an SMT processor. Since the EV8 can resolve 16 branches per clock cycle, using a two-level branch prediction scheme results in extremely complex hardware implementation.

The authors state that two ports, one for each instruction fetch block, are necessary for the branch history table. One read can then produce eight sequential history entries, one for each instruction in the fetch block. The difficulty is that sixteen pattern table reads now must occur, which is not possible given the timing constraints of the EV8. A global scheme, on the other hand, only requires the two ports to the pattern table. The authors also claim (without data) that global prediction schemes are preferable for SMT processors.

In [7] Hily et al. evaluate the bimodal, gshare, and gselect predictors on a 4 context SMT for multiprogrammed and multithreaded workloads. Their results show that gshare typically outperforms gselect, which in turn outperforms the bimodal predictor for all simulated workloads.

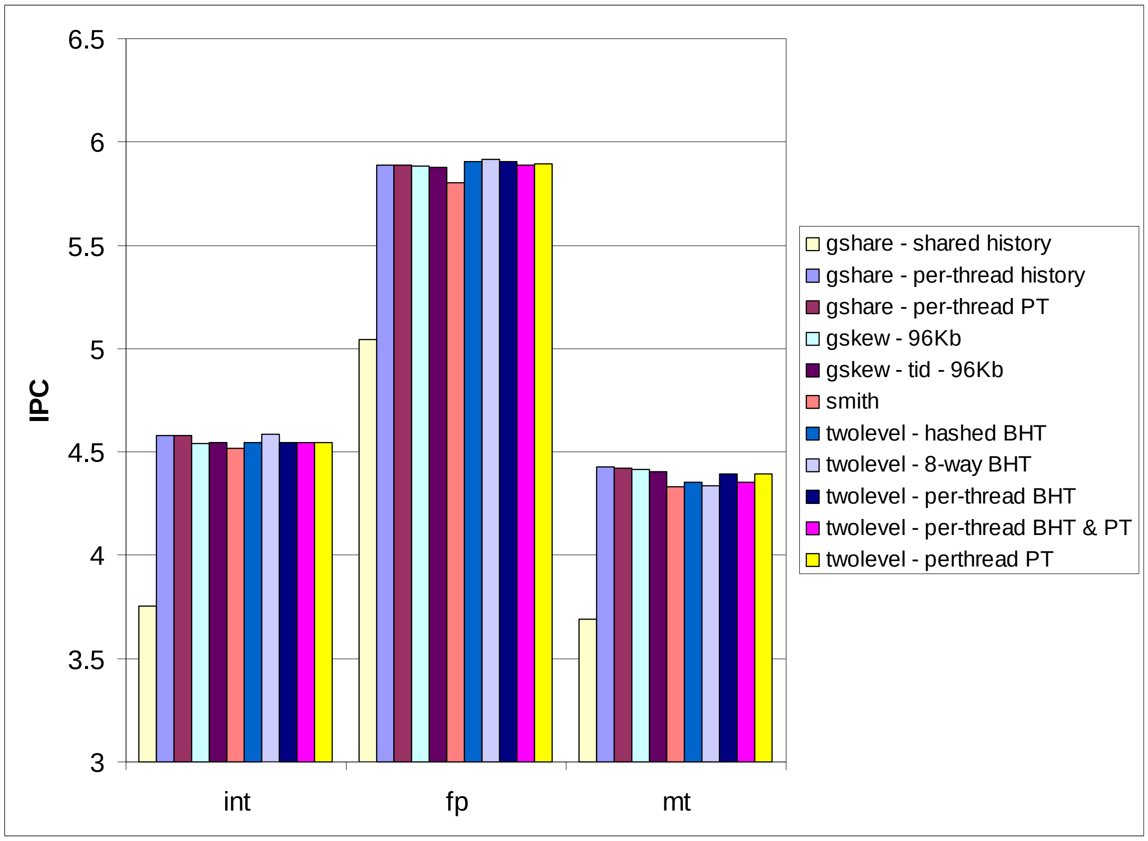

Figure 7: IPC for 4 simultaneous threads.

7 Conclusions

In this paper we have discussed the issues with branch prediction in processors with simultaneous multithreading (SMT). We have demonstrated the importance of using global history to make predictions, and have shown that global histories must be stored for each thread context to avoid a disastrous impact to prediction accuracy and throughput. Evaluations of per-thread predictor tables have shown that the reduction in table size caused by separating them for each thread outweighs any benefit from eliminating aliasing.

In this work we have validated the result that SMT greatly reduces the dependence of throughput on branch prediction accuracy. However, branch prediction accuracy is still important. We have shown that when 1 thread is running on an SMT, performance is still strongly dependent on branch prediction. The implementation difficulties of two-level branch predictors in an SMT processor, combined with our results showing the superior performance of global predictors, provide a powerful argument for using a branch prediction scheme based on global history.

Bibliography

[1] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In Proceedings of the 23rd Annual International Symposium on Computer Architecture, pages 191--202, May 22--24, 1996.
[2] Andre Seznec, Stephen Felix, Venkata Krishnan, and Yieannakis Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In Proceedings of the 29th Annual Symposium on Computer Architecture, May 25-29, 2002.
[3] J. E. Smith. A study of branch prediction strategies. In Proceedings of the 8th Annual Symposium on Computer Architecture, pp. 135-148, May 1981.
[4] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In Proceedings of the 24th Annual Symposium on Microarchitecure.
[5] S. McFarling. Combining branch predictors. In DEC WRL TN-36, June 1993.
[6] Pierre Michaud, Andre Seznec, and Richard Uhlig, Skewed branch predictors, Tech. report, IRISA publ. int. 1031, June 1996.
[7] S. Hily and A. Seznec. Branch Prediction and Simultaneous Multithreading. In International Conference on Parallel Architecture and Compilation Techniques, 1996.
[8] Steve Swanson, Luke McDowell, Michael Swift, Susan Eggers, Henry Levy. An Evaluation of Speculative Instruction Execution on Simultaneous Multithreaded Processors. January 2002. Submitted for publication. http://www.cs.washington.edu/research/smt/papers/smtspeculation.pdf.