Low-cost Techniques for Reducing Branch Context Pollution in a Soft Real-time Embedded Multithreaded Processor

Emre Özer, Alastair Reid and Stuart Biles
ARM Ltd., Cambridge, UK
{emre.ozer, alastair.reid, stuart.biles}@arm.com

Abstract

In this paper, we propose two low-cost and novel branch history buffer handling schemes aiming at skewing the branch prediction accuracy in favor of a real-time thread for a soft real-time embedded multithreaded processor. The processor core accommodates two running threads, one with the highest priority and the other thread is a background thread, and both threads share the branch predictor. The first scheme uses a 3-bit branch history buffer in which the highest priority thread uses the most significant 2 bits to change the prediction state while the background thread uses only the least significant 2 bits. The second scheme uses the shared 2-bit branch history buffer that implements integer updates for the highest priority thread but fractional updates for the background thread in order to achieve relatively higher prediction accuracy in the highest priority thread. The low cost nature of these two schemes, particularly in the second scheme, makes them attractive with moderate improvement in the performance of the highest priority thread.

1. Introduction

Modern processors use predictors like bimodal, local, global or hybrid branch predictors. The common structure in all of these predictors is the 2-bit branch history buffer that hints the direction of branches. When a branch instruction is encountered, the direction prediction is made by looking at the counter value corresponding to the selected branch history buffer (BHB) entry.

In a multithreaded (MT) processor core, the BHB is shared among threads as are many other resources within the core. When a BHB entry is accessed by more than one thread, the branch predictor performance can degrade for one thread in the best case but in all threads in the worst case. For instance, consider the case where one thread keeps getting correct taken branch predictions and incrementing the counter while another keeps decrementing the same counter for correct not-taken branch predictions. In this case, the branch prediction accuracies for both threads drop.

Sharing the BHB blindly can be tolerable in a multithreaded processor where all threads have equal priorities. However, the situation can be unacceptable in a soft real-time embedded multithreaded environment where one thread is the highest priority (HP) or real-time thread while the others are non-critical low-priority (LP) threads. The real-time task or thread in a soft real-time system is not time critical, which means that no catastrophic event occurs when the real-time task misses its deadline but some form of quality is lost, e.g. real-time video conferencing. The conflicts in the BHB entries may degrade the performance of the HP thread. Ideally, the branch prediction accuracy should be skewed in favor of the HP thread at a cost of reduction in the branch prediction accuracies of the LP threads. In this way, the prediction accuracy of the HP thread can be maintained at a desired level while providing acceptable prediction accuracies for the LP threads.

The ideal prediction accuracy for all threads in a multithreaded processor can be obtained by replicating the branch predictor so that each thread has its own 2-bit BHB. This provides the optimal HP thread branch prediction accuracy but at the highest hardware cost. At the other end of the spectrum, we have a shared 2-bit BHB for all threads. This represents the worst HP thread branch prediction accuracy but at the lowest hardware cost. Because of power and cost budget, embedded processors avoid using extensive hardware, so sharing the BHB seems more reasonable.

In this paper, we have an embedded soft real-time multithreaded processor having one highest priority thread and one or many low priority threads, and all threads share a branch predictor. We propose two schemes operating on a shared BHB, and they skew the branch prediction accuracy towards the highest priority thread while using small extra hardware. The

first scheme uses a 3-bit BHB in which the HP thread uses the most significant 2 bits to change the prediction state (i.e. inc/decrement by 2), and the LP thread(s) use the least significant 2 bits to change the prediction state (i.e. traditional inc/decrement by 1). Sharing the 3-bit BHB provides a skewed branch predictor that improves the branch prediction accuracy in favor of the HP thread when BHB collisions occur. The second scheme, also called the fractional counter update mechanism, uses the traditional shared 2-bit BHB that skews the branch prediction towards the HP thread by doing regular increment/decrements for HP thread but fractional increment/decrements for LP threads.

The main motivation of this paper is to come up with **efficient and low-cost branch prediction schemes** that can improve the performance of a real time thread in a soft real-time embedded multithreaded processor. The required extra hardware is considerably simple particularly in the fractional counter update scheme where only an LFSR-based pseudo random number generator and a couple of logic gates suffice.

The organization of the paper is as follows: *Section 2* gives background on the branch prediction of single and multi-threaded processors. *Section 3* introduces the two low-cost and novel BHB handling schemes. *Section 4* presents the experimental framework and results. *Section 5* discusses the related work. Finally, *Section 6* concludes the paper with future work.

2. Background

2.1. Branch Prediction in Single-threaded Processor

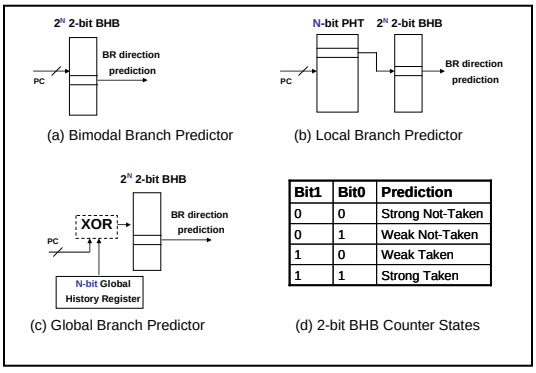


Figure 1 2-bit BHB arrays in different branch predictors

Figure 1 shows the BHB counters used in several branch predictors [2 and 11]. In a bimodal branch predictor (a), the prediction is made by accessing the 2-bit BHB array by the PC address of the branch instruction. For the local branch predictor (b), the PC of the branch instruction accesses a local history of the branch in a pattern history table (PHT). The local branch history is kept as a shift register that is shifted by 1 if the branch is taken or shifted by 0 otherwise. The 2-bit BHB entry is accessed by this local branch history value. In the case of a global branch predictor (c), the branch history is collected globally by a global history register (GHR). The 2-bit BHB counter is accessed by either concatenating the GHR content to PC or XORing it with PC. Once the 2-bit BHB is read, the branch prediction direction can be made using the 2-bit BHB counter state table (d). Later on, if a branch is correctly predicted, the 2-bit BHB counter is incremented by 1 towards strong taken state. If it is mispredicted, the counter is decremented by 1 towards strong not-taken state.

There are also more complicated branch predictors such as [14 and 15]. These techniques take advantage of multiple different branch predictor tables to improve the prediction accuracy.

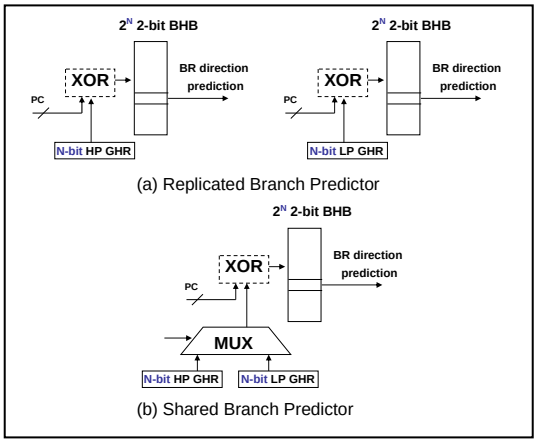


Figure 2 Branch prediction in an MT processor

2.1. Branch Prediction in Multi-threaded Processor

For an MT processor core, the branch predictor can be either replicated for each thread or shared among all threads as shown in Figure 2.

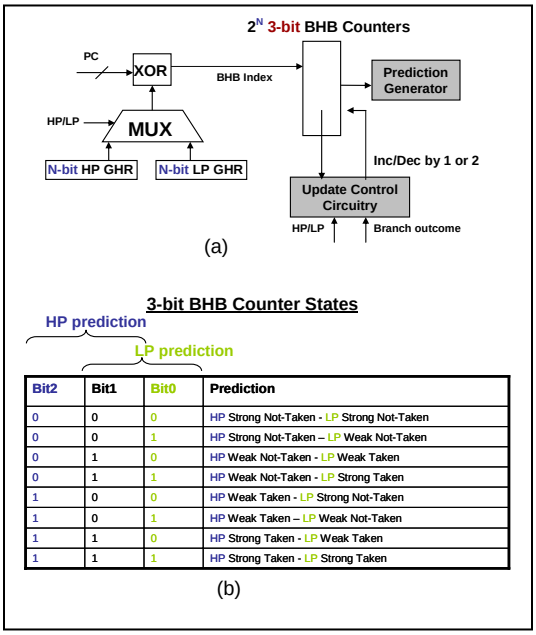
For the replicated branch predictor (a), both GHR and BHB are replicated. It gives the best branch prediction performance because there will not be any inter-thread collisions in the BHB array, but this will happen at the highest hardware cost. On the other hand, the shared branch predictor (b) replicates only the GHR but shares the long BHB array among all threads. So, each thread tracks its own global history. This is the least costly option but inter-thread

collisions can occur in the shared BHB array, which may potentially degrade the branch prediction accuracy of the threads.

For an MT processor core in which all threads have equal priority for using all processor resources, the branch prediction accuracy loss because of sharing the BHB will most likely be similar for all threads. This can be acceptable as no thread has higher priority than another. On the other hand, the situation is somewhat different in a real-time MT processor in which one thread has higher priority to use processor resources than other threads, and also the BHB is shared among all threads. An equal amount of loss in branch prediction accuracy in every thread cannot be acceptable in such a system. Thus, a shared BHB that is slightly skewed to provide higher real-time thread branch prediction accuracy than the other low-priority threads would be more appropriate for a real-time MT processor.

3. Branch Prediction in Soft Real-time Multi-threaded Processor

We propose 2 different schemes that can skew the branch prediction accuracy towards the HP thread.



Bit2	Bit1	Bit0	Prediction
0	0	0	HP Strong Not-Taken - LP Strong Not-Taken
0	0	1	HP Strong Not-Taken – LP Weak Not-Taken
0	1	0	HP Weak Not-Taken - LP Weak Taken
0	1	1	HP Weak Not-Taken - LP Strong Taken
1	0	0	HP Weak Taken - LP Strong Not-Taken
1	0	1	HP Weak Taken – LP Weak Not-Taken
1	1	0	HP Strong Taken - LP Weak Taken
1	1	1	HP Strong Taken - LP Strong Taken

Figure 3 3-bit BHB scheme

3.1 Shared 3-bit BHB Scheme

The 3-bit BHB scheme skews the branch prediction accuracy in favor of HP thread so that higher HP prediction accuracy than the LP thread(s) can be obtained. Figure 3 shows the 3-bit BHB scheme (a) as well as the prediction state table (b). Although the figure shows only 2 threads, the scheme can support one HP and many LP threads using the same 3-bit BHB array.

The first two bits in 3-bit BHB entry determine the branch prediction direction of the LP thread(s) while the last two bits determine the branch prediction direction of the HP thread. Bit1 is shared between HP and LP threads. When the LP thread makes a prediction, it reads only the first 2 bits from the BHB entry. Similarly, when the HP thread makes a prediction, it reads only the last 2 bits from the BHB entry.

Upon branch execution, if the LP thread branch prediction is correct, then the BHB counter is incremented by 1. If there is a misprediction, the counter is decremented by 1. This is exactly the same as the 2-bit traditional BHB scheme. On the other hand, the BHB counter is incremented by 2 if the HP thread branch prediction is correct. If there is a misprediction, the counter is decremented by 2. The increment or decrement is performed by 2 because we only care about the last 2 bits for the HP thread[1].

This 3-bit BHB organization skews the branch prediction accuracy in favor of the HP thread because it requires two successive LP increments or decrements in the same BHB entry in order to corrupt the HP branch prediction state. For instance, let us assume that the current counter value is 100 (i.e. “HP Weak Taken” and “LP Strong Not-taken”) in a BHB entry accessed by the LP thread that makes a prediction. It will predict “Not-taken”. Later upon execution, let us assume that it mispredicts (i.e. the branch is actually “Taken”). Now, the counter in the same BHB entry is accessed again and incremented by 1 and the new counter value becomes 101 (i.e. “HP Weak Taken” and “LP Weak Not-taken”). Note that this change in LP branch prediction state does not modify the HP branch prediction state, so the HP state is not affected. It will only change if the LP thread accesses the same BHB entry and mispredicts the branch. On the contrary, if the HP thread increments the counter value (i.e. 100) by 2 on a correct prediction, the new counter value would be 110 (i.e. “HP Strong Taken” and “LP Weak Taken”). The new counter value implies that the LP branch prediction state changes from “Strong Not-taken” to “Weak Taken” in one step. Thus, a change in HP prediction state can change the LP state immediately in the opposite direction. Because of this behavior, the branch predictor is skewed to provide

[1] The counters are saturating, so decrementing a counter having a value of 1 by 2 results in 0.

better HP thread prediction accuracies than the LP ones.

3.2. Shared 2-bit BHB with Fractional Counter Update Scheme

Figure 4 shows the view of the shared 2-bit BHB with fractional counter update mechanism. Each thread makes a branch direction prediction as shown in Figure 4**a** by accessing the BHB and reading the counter value to the prediction generator that makes a final decision on the prediction.

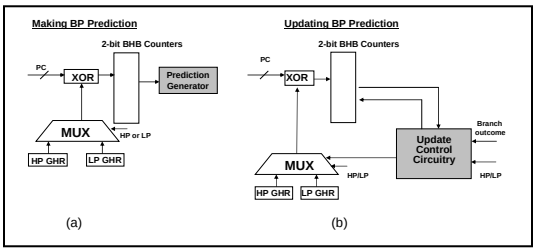


Figure 4 Shared 2-bit BHB with fractional update scheme

Figure 4**b** shows how the BHB is updated. The HP thread can update the BHB in a conventional way, i.e. accessing the associated counter, reading it, incrementing or decrementing depending on the outcome of the branch, and finally writing it back to the same BHB entry. However, the LP thread increments/decrements the BHB counters by fractional values rather than integer value of 1. The motivation behind the fractional updating is exactly the same as the shared 3-bit BHB scheme, i.e. skewing the branch prediction accuracy in favor of the HP thread. The branch prediction state information belonging to the HP thread can be maintained for a certain period of time when the LP thread fractionally changes its branch prediction state information when the same BHB entry is accessed by the two threads. The main idea is also similar to the shared 3-bit BHB scheme in that the HP thread updates the BHB by larger values than the LP thread. However, the advantage of the fractional updating over the shared 3-bit BHB is that the fractional update can do this on the traditional shared 2-bit BHB. The update control unit in the figure is responsible for incrementing and decrementing by an integer of 1 for HP and by fractions for LP thread.

The details of the update control unit are shown in *Figure 5*. Basically, it consists of the counter update control and random number generator. Incrementing/decrementing by a fractional number can be implemented probabilistically by using a random number generator [1]. For instance, it is sufficient for implementing a fractional update by ½ if the random number generator generates either 0 or 1. If the random number is 0, the counter is updated (i.e. increment or decrement) by 1. If it is 1, then the counter is not changed. Statistically, this approximates the behavior of updating the counter by ½. Updating by an arbitrary fraction can be implemented similarly as such: for a fraction of 1/N, a random number generator generates an integer number between 0 and N-1. If the generated random number is 0, then the counter is updated by 1. For any other value than 0, the counter is not changed. In fact, this can be even taken further as such that different fractions can be assigned to branch types. For instance, branch type A increment/decrements by 1/4 while branch type B increment/decrements by 1/8. In this paper, we did not explore this, i.e. all branch types change the BHB counters by the same fraction.

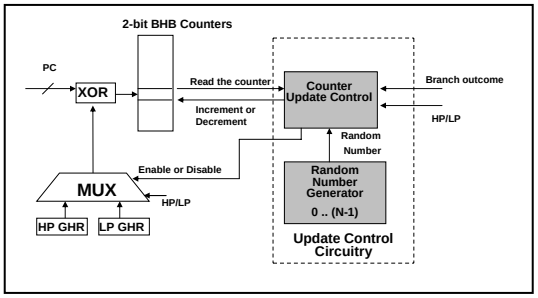


Figure 5 Update control unit in the fractional update scheme

The counter update control takes the thread type and/or random number as input. If the HP thread needs to update the BHB, then the random number generator is not enabled and thus no random number is generated. In that case, the counter update control enables the MUX so that the GHR of the HP thread is forwarded to the XOR to access the BHB, then reads the counter value and finally writes it back by incrementing or decrementing by an integer value of 1. If the LP thread needs to update the BHB, the random number generator is enabled and generates a random number between 0 and (N-1). The random number is sent to the counter update control that first enables the MUX so that the GHR of the LP thread is forwarded to the XOR to access the BHB. Later, the counter is read and incremented/decremented by 1 and written back to the same entry. The counter update control disables the MUX if the random number is greater than 0. In this case, the access to the BHB by the LP thread is blocked by not forwarding its GHR to the XOR.

An implementation of the shared 2-bit BHB scheme with a fraction of ¼ is shown in Figure 6. The random

number generator is implemented by a linear feedback shift register (LFSR) that generates numbers between 0 and 3. The hardware cost of this particular implementation is minimal, i.e. an LFSR generating 2-bit random number, a NOR and an OR gates. The NOR and OR gates are used to enable or disable the MUX if the BHB access is done by the LP thread. Only when the generated random number is 0 and the current thread is LP (i.e. thread id is 0), the MUX is enabled to forward the GHR of the LP thread to access the BHB. For any other generated random number for the LP thread, the MUX is disabled. Also, if the BHB is accessed by the HP thread, the MUX is always enabled.

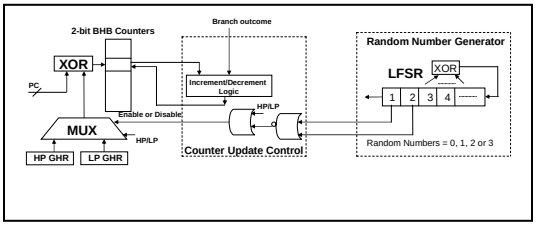


Figure 6 An implementation of the shared 2-bit BHB scheme with a fraction of ¼

Table 1 Simulated processor and memory model parameters

Parameters	Details
Processor type	In-order superscalar
Issue width	Dual-issue
Fetch bandwidth	2 32-bit or 4 16-bit instruction fetch per cycle
Decode bandwidth	2 instructions per cycle
# of Threads	2
On-chip L1 Instruction Cache	4-way 32KB with 1-cycle hit time
On-chip L1 Data Cache	4-way 32KB with 1-cycle hit time
On-chip L2 Unified I & D Cache	8-way 256KB with 8-cycle hit time
Memory Access Latency	60 cycles
Instruction and Data TLB size	32-entry fully-associative
Branch Predictor	Global Branch Predictor with *shared* BHB and *replicated* Global Branch History Register
Return Address Stack	*Replicated* 8-entry

4. Experimental Evaluation

We have performed a cycle-accurate simulation of a Simultaneous Multithreading (SMT) [4] implementation of an ARMv7 architecture-compliant ARM processor core. The processor is modeled as a dual-thread soft real-time SMT core in which one thread has the highest priority over the other one. The HP thread has priority using the fetch, decode and issue resources over the LP thread in the SMT model. The processor core and memory model parameters are shown in Table 1.

We have used 14 benchmarks from the *EEMBC* benchmark suite [16] covering a wide range of embedded applications including consumer, automotive, telecommunications and DSP. We run all possible dual-thread permutations of these benchmarks. A dual-thread simulation run completes when the HP thread finishes its execution, and then we collect statistics such as total IPC, degree of LP thread progress, HP thread speedup, branch prediction accuracy and etc.

In order to observe a high number of collisions in the BHB, we choose a small shared BHB of 16 entries to stress it out aggressively as the *EEMBC* benchmarks are, in general, kernel programs that do not put pressure on relatively large BHBs. Although a 16-entry BHB is too small and unrealistic for a commercial processor, the experimental results with 16-entry BHB for small benchmarks can reasonably approximate a system with a more realistic BHB (e.g. 4096-entry) for large and realistic applications. Also, the 512-entry branch target buffer (BTB) is replicated for each thread to isolate its effects on the branch direction prediction.

We compare the shared 3-bit BHB and shared 2-bit with fractional counter update schemes to the upper and lower-bound models. Replicated 2-bit BHB scheme that gives the highest performance for both threads is the upper-bound model while traditional shared 2-bit BHB scheme that gives the lowest performance for HP thread becomes the lower-bound model. Also, we use 4 different fractions 0.5, 0.25, 0.125 and 0.0625 for the fractional counter update scheme.

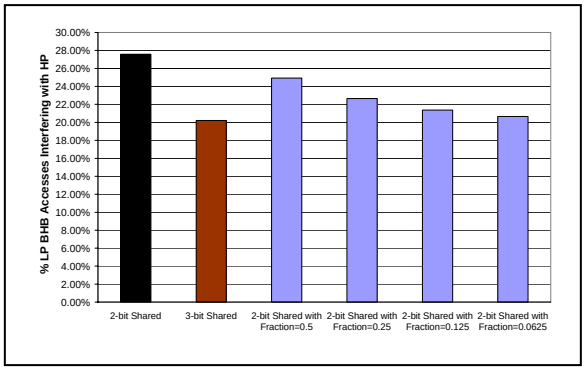


Figure 7 Percentage of LP BHB accesses interfering with the HP branch prediction state

Figure 7 shows the percentages of the collisions or aliases in the BHB by the LP thread interfering with the HP thread branch prediction state. Since the replicated branch predictor does not exhibit any BHB aliases, it is not shown in the graph. About 28% of all LP BHB accesses interfere with the HP thread's branch prediction state. The shared 3-bit BHB scheme drops this rate to 20.2% by taking advantage of extra bit in each BHB counter. Similarly, the shared 2-bit BHB with fractional update scheme approximates the behavior of the shared 3-bit BHB scheme using the shared 2-bit BHB. The alias rate drops from 25% with a fraction of 0.5 to 20.7% with a fraction of 0.0625. As the fraction gets smaller, the probability of interfering with HP thread branch prediction state diminishes. Thus, we would expect that the performance of the HP thread should improve considerably.

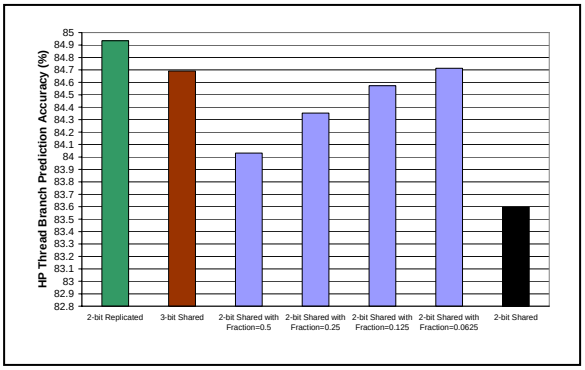


Figure 8 HP thread branch prediction accuracy

Figure 8 shows the branch prediction accuracy of the HP thread for all schemes including the replicated 2-bit BHB. The replicated and shared 2-bit BHB schemes represent the two extreme ends of the prediction accuracy spectrum. Ideally, the HP branch prediction accuracy should be as close to the replicated 2-bit BHB scheme as possible. Obviously, the HP prediction accuracies of the shared 3-bit BHB and shared 2-bit BHB with fractional counter update schemes are all better than the shared 2-bit BHB as the disturbance by the LP thread is reduced. The HP prediction accuracy improves as the fraction amount gets smaller tracking the behavior mentioned when describing Figure 7. For instance, the accuracy becomes 84.71% for a fraction of 0.0625, which is slightly better than the shared 3-bit BHB scheme (i.e. 84.7%). The best case (i.e. the replicated 2-bit BHB) provides 85% prediction accuracy while the worst case (i.e. the shared 2-bit BHB) has a prediction accuracy of 83.6%. The shared 3-bit scheme boosts the prediction accuracy to 84.7%, which is very close to the best case. As expected, the prediction accuracy increases as the fractions gets smaller in the shared 2-bit BHB with fractional counter update scheme. In fact, the prediction accuracy for a fraction of 0.0625 is slightly better than the prediction accuracy of the 3-bit shared BHB scheme.

The improvement in the HP thread branch prediction accuracy comes at a cost of lower branch prediction accuracy in the LP thread. Figure 9 presents the branch prediction accuracies for the LP thread. For the LP thread, both the replicated and shared 2-bit schemes outdo the 3-bit shared BHB and the shared 2-bit BHB with fractional update schemes. This is somewhat expected because these two schemes treat the LP thread as the sacrificial thread, meaning that some performance loss in the LP thread is acceptable as long as this loss in the LP thread can be used to ramp up the performance of the HP thread.

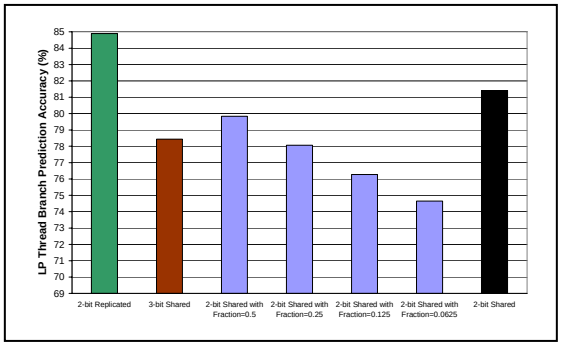


Figure 9 LP thread branch prediction accuracy

Figure 10 shows the slowdown in the HP thread's execution time relative to the execution time of the best case (i.e. replicated 2-bit BHB). The lower the height of the bar is, the better the scheme's performance. The HP thread in the worst case (i.e. 2-bit shared BHB) slows down by slightly more than 0.9%. This drops to slightly less than 0.18% for the 3-bit shared BHB scheme due to its non-interfering way of using the BHB table in favor of the HP thread. For the shared 2-bit BHB with fractional counter update scheme, the slowdown in the HP thread can be reduced up to 0.14% with a fraction of 0.0625, and can even be reduced further if smaller fractions are used. Although the improvement in the execution time of the HP thread using the novel schemes is small relative the shared 2-bit BHB scheme, the hardware overhead to achieve this improvement is also small, particularly in the case of the 2-bit shared BHB with fractional counter update scheme.

Table 2 provides the cycles per instruction (CPI) of the LP thread to show how big the performance sacrifice in the LP thread as it is the sacrificial thread.

The CPI of the LP thread shows how much the LP thread progresses under the shadow of the HP thread. As in Figure 9, the replicated and shared 2-bit BHB schemes outperform the two novel schemes for the LP thread. So, the performance loss in the LP thread is about 0.9% for the 3-bit shared scheme. This varies from 0.7% to 4.7% for the 2-bit shared BHB with fractional counter update scheme with fractions varying from 0.5 to 0.0625.

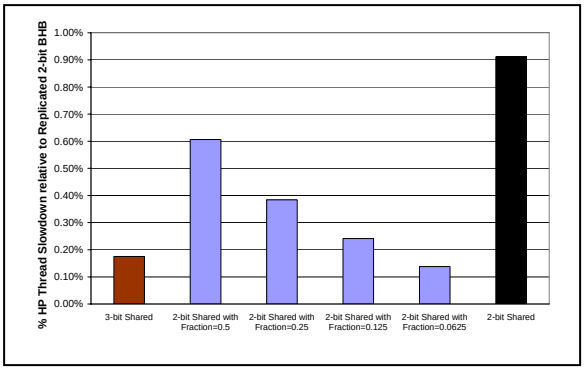


Figure 10 Percentage slowdown in HP thread execution time relative to the replicated 2-bit BHB scheme

Table 2 CPI of the LP thread

BHB Scheme	LP CPI
Replicated 2-bit	4.18
Shared 2-bit	4.27
Shared 3-bit	4.31
Shared 2-bit with Fraction=0.5	4.30
Shared 2-bit with Fraction=0.25	4.35
Shared 2-bit with Fraction=0.125	4.41
Shared 2-bit with Fraction=0.0625	4.47

5. Related Work

The effects of various types of shared and replicated branch predictors including the destructive and constructive interferences in the prediction tables on an SMT processor core have been extensively investigated by *Hily et al.* [3], *Ramsay et al.* [12] and *Seznec et al.* [13]. However, they do not discuss any efficient use of the shared branch prediction tables among threads such as skewing branch prediction accuracy in the presence of strict thread priority.

In generic SMT processor models such as [4, 5 and 6], the main focus is to improve the overall processor throughput, and the BHB array is shared and updated as in the single-threaded processor core. In certain SMT processors where one thread can have a certain priority over the others such as in [7, 8, 9 and 10], the BHB is shared without differentiating the priority of the threads.

6. Conclusion and Future Work

We have shown that the shared skewed 3-bit BHB scheme can outperform the shared 2-bit BHB scheme for the HP thread and is very close to the performance of the replicated 2-bit BHB in a soft real-time embedded multithreaded processor core. The performance improvement of the HP thread comes at a cost requiring one extra bit in each BHB entry.

On the other hand, the 2-bit shared BHB with fractional counter update scheme does not change the hardware structure of the BHB but introduces a very simple probabilistic BHB update facility. As the update fractions decrease, the HP thread prediction accuracy increases, and the HP thread speeds up. However, the LP prediction accuracy in this scheme falls below the LP prediction accuracy of the traditional shared 2-bit scheme as the probability decreases. This, in turn, leads to a low instruction throughput (i.e. CPI) of the LP thread. The decision about which fractional value is the best depends on the objectives to meet. If the objective is to maximize the HP thread execution time, then 1/16 or lower probabilities should be selected. On the other hand, higher fractions (e.g. ¼) must be used if a balance among HP execution time, LP thread performance and total instruction throughput are sought.

We are planning to extend the concept of probabilistic fractional counter updating to different branch types such as direct, indirect branches and etc. in the context of a single-threaded processor core. When different branch type instructions access the same BHB entry, a collision can occur. A collision may corrupt the current branch prediction state when two collided different branch type instructions go in different directions. This issue can be alleviated by allowing some branch type instructions to update the BHB entries by fractional increments and decrements. For instance, a direct branch type is more common than an indirect one, so an indirect branch may increment/decrement by fractions.

7. References

[1] R. Morris, “Counting large numbers of events in small registers”, *Communications of the ACM*, Volume 21, Issue 10, October 1978.

[2] S. McFarling, "Combining Branch Predictors", *Tech. Rep. TN36*, Digital Equipment Corporation, June 1993.

[3] S. Hily and A. Seznec, "Branch Prediction and Simultaneous Multithreading", *Proceedings of Parallel Architectures and Compilation Techniques*, 1996.
[4] D. M. Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism", *International Symp. on Computer Architecture (ISCA)*, 1995.
[5] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading Technology Architecture and Microarchitecture", *Intel Technology Journal, 3(1)*, Feb. 2002.
[6] F. N. Eskesen, M. Hack, T. Kimbrel, M. S. Squillante, R. J. Eickemeyer and S. R. Kunkel, "Performance Analysis of Simultaneous Multithreading in a PowerPC-based Processor", *The Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD'02) held in conjunction with the International Symposium on Computer Architecture (ISCA)*, June 2002.
[7] S. E. Raasch and S. K. Reinhardt, "Applications of Thread Prioritization in SMT Processors", *Proceedings of Multithreaded Execution, Architecture and Compilation Workshop*, January 1999.
[8] G. K. Dorai and D. Yeung, "Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance", *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, 2002.
[9] F. J. Cazorla, P. M.W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, M. Valero, "Predictable performance in SMT processors", *Proceedings of the 1st Conference on Computing Frontiers*, April 2004.
[10] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer and J. B. Joyner "POWER5 System Microarchitecture", *IBM J. Research and Development*, Vol. 49 No. 4/5 July/September 2005.
[11] T. Y. Yeh and Y. N. Patt. "Two-level Adaptive Training Branch Prediction", *Proceedings of the 24th Annual Workshop on Microprogramming* (MICRO-24), Dec. 1991.
[12] M. Ramsay, C. Feucht, and M. H. Lipasti, "Exploring Efficient SMT Branch Predictor Design", *Workshop on Complexity-Effective Design*, in conjunction with ISCA, June 2003.
[13] A. Seznec, S. Felix, V.Krishnan and Y. Sazeides, "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor", *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.
[14] P. Michaud, A. Seznec, and R. Uhlig, "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors", *Proceedings of the 24th Annual International Symposium on Computer Architecture* (ISCA-97), June 1997.
[15] A. N. Eden and T. Mudge, "The YAGS Branch Prediction Scheme", *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.
[16] http://www.eembc.org/