Effect of CGMT Processors on Branch Prediction

Xidong Wang, Ning Li
Computer Science Department
University of Wisconsin
Madison, WI 53706
{wxd, ning}@cs.wisc.edu

Abstract

CGMT can exert significant influence on the design and implementation of branch prediction. In this paper, the design of several typical branch predictors such as bimodel, GShare and McFarling is examined and branch prediction performance is improved by making some decisions with the help of TPC-W benchmark concerning whether to share some data structures among multiple threads or not. For a typical two-level adaptive branch predictor such as GShare, RAS & BHR should be kept thread private, BHT & BTB should be shared among threads. Also we make special effort to improve BTB hit rate besides keeping it shared. Algorithm called victim BTB with similar thought of victim cache is presented and a couple of variant algorithm, inter-thread victim BTB and intra-thread victim BTB are described in detail. Experimental results show that BTB hit rate is improved considerably when BTB size is not very large.

1. Introduction

Multithreading processor can achieve significant performance gain by overlapping various threads executions. In Coarse Grained Multithreading (CGMT) processor, multiple thread contexts exist within each physical processor, and thread-switch logic is used to swap threads in and out when long latency events stall execution of a particular thread. A long latency event could be that a cache miss has occurred in the primary thread, or a primary thread has entered an idle loop or is spinning on a lock, etc.

However, the overlapping way of execution can exert undesirable effect on branch prediction performance. In case of GShare predictor, if both branch history register (BHR) and branch history table (BHT) are shared among multiple threads, experimental results provided by TPC-W paper show that serious performance degradation can be introduced. Although TPC-W paper pointed out that a private
BHR for each thread and one shared BHT can improve branch prediction accuracy, much more work should be exploited in order to figure out the influence that multithreaded execution pattern exert on sharing attributes of each part of branch prediction data structures.

In this paper, we focus on the sharing pattern of data structures of various branch predictors, such as BHR, BHT, RAS (return address stack) and BTB (branch target buffer). Multithreading may have varying impact on these data structures because they rely on different architecture aspects. Data structures of three kinds of predictors are examined and compared -- the simple Bimodel predictor, the GShare predictor and the complex McFarling predictor. Most data structures have demonstrated similar sharing attributes among different predictors. Sharing pattern of data structure in typical two-level adaptive branch predictor is described in detail. Below is the design of a typical two-level adaptive branch predictor.

![Figure 1: design of typical two-level adaptive branch predictor](image)

The paper is organized as follows. Section 2 will give an overview of the simulation environment. Section 3 will examine BHR, BHT, RAS data structures of GShare, Bimodel, McFarling branch predictors. Section 4 will characterize BTB performance of GShare predictor, and present the improvement algorithm victim BTB, then evaluate its effectiveness. Related work is discussed in section 5 and we draw our conclusions in section 6.
2. Methodology

2.1 Simulation Environment

As in TPC-W paper [Cain], we build our simulation environment based on an augmented version of SimOS-PPC full system simulator, which runs a slightly modified version of AIX 4.3.1. The processor model approximates the behavior of the RS64-III (Pulsar) processors used in IBM RS/6000 S80 Systems. The configurations could be one thread per processor, two threads per processor and four threads per processor. A four-state state machine is adopted to switch threads, with coarse-grain thread switch logic. Forward progress is guaranteed and a 3-cycle thread switch penalty is assumed. Also an aggressive 4-way snooping-bus shared memory is applied.

2.2 Workload

Shopping mix of the TPC-W web benchmark is taken as primary workload to measure the branch prediction performance on multithreading. The TPC-W is a transactional benchmark, which models an on-line bookstore. The environment of this benchmark could be depicted as multiple concurrent on-line browser sessions, web serving of web pages, and a variety of access and update patterns to a database. The specification defines three different mixes of web interactions, and the shopping mix is the primary mix that is used, which reflects an average shopping scenario. All of the TPC-W application logic is implemented in Java using the Java Servlet API.

3. Simulation Results and Analysis

3.1 BHR & BHT results and analysis

In a typical two-level adaptive branch predictor such as GShare, BHR (Branch History Register) stores branch history information related to PC address. BHT (Branch Pattern History Table) stores branch history information related to some history pattern. They cooperate together to predict the branch direction (taken or not taken). Branch pattern history information is identical among threads, and shared BHT owns more
entries than private BHT, therefore conflicts in BHT are minimized and branch prediction performance is improved.

BHR situation is not so easy to make decision whether to share or not. Different thread can produce different PC-related branch history for they usually execute different code segment. If BHR is shared, inter-thread conflicts on BHR will increase greatly for multithread interference. On the other hand, intra-thread conflicts on BHR are reduced for larger size than private BHR. But experimental data below shows that size benefit cannot offset the performance loss caused by multithread interference. For 2- or 4-thread, when BHR is shared and size of BHR is increased, the branch performance does not benefit too much. 4-thread situation even gets slightly worse performance sometime. The reason we provide is that change in multithread interference offsets the size increase benefit. In conclusion, it suggests that private BHR can perform better than shared BHR.

Private BHR and shared BHT can be integrated together to achieve best possible result. We can see from data below that that algorithm perform relatively better than solely shared or private BHR and BHT algorithm.

One thing which should be noticed here is that the experimental data and conclusion are different from those presented by TPC-W paper [3]. Actually we found some implementation bugs in branch predictor of TPC-W version. The most important bug fix we made to TPC-W branch predictor is, using some middle bits of PC instead of lower bits to index BHR. Because the lowest two bits of PC are always zero in simulation environment, so it is not reasonable to use them as index.

![Figure 2: multithreading effect on BHR & BHT of GShare predictor](image_url)
3.2 Bimodel & McFarling branch predictors

![Graph showing the performance of Bimodel and McFarling predictors.](image)

Figure 3: multithreading effect on Bimodel & McFarling predictors

The results in Figure 3 shows that shared Bimodel branch predictor outperforms private Bimodel branch predictor, while private McFarling branch predictor has a higher prediction accuracy than its shared counterpart, similar to GShare branch predictor. Bimodel branch predictor makes prediction based on only the current PC address, without utilizing any history of previous branches. So inter-thread interferences have less negative impact on the performance of Bimodel branch predictor. Shared Bimodel branch predictor is thus able to take advantage of a larger size table without suffering much from inter-thread interferences, consequently beats the performance in the private case.

McFarling branch predictor makes use of Bimodel and GShare branch predictors, estimates which predictor is more accurate and then predicts. Its sharing attribute really depends on how much it uses each of the two predictors. Because private GShare predictor predicts more precise than private Bimodel predictor, private McFarling uses more of private GShare predictor and outperforms shared McFarling predictor, which uses more of shared Bimodel predictor. That also says why the difference of shared and private McFarling predictors are not as significant as for GShare predictor.

3.3 RAS results & analysis

RAS (return address stack) is a stack data structure a thread uses to save return address of procedure calls. Return address information of procedure calls can rarely be identical among threads, unless multithreads are executing same code segment and their progress in execution is also the same. Therefore
return address information can hardly be shared among threads. It is straightforward to keep RAS data structure thread-private so that procedure call returns of different threads do not interfere together.

![Graph showing mispredictions vs. number of RAS entries for different thread configurations](image)

Figure 4: multithreading effect on RAS of GShare predictor

From the figure above, we can see that when RAS is shared among multithreads, the RAS prediction performance is degraded greatly. 4-thread degradation is worse than 2-thread since severe inter-thread RAS interference is introduced from more concurrent threads execution. When private RAS is made use, size of RAS owned by each thread is smaller compared with shared RAS. That will degrade RAS performance somehow. However, multithread interference is removed and RAS performance will be improved greatly. The overall effect of private RAS is far better than shared RAS.

### 3.5 BTB results & analysis

BTB hit rate is another interesting issue when considering the effect of CGMT on branch prediction. Individual threads can share one BTB buffer or they each can be assigned with private BTB. Given fixed size of BTB, sharing option can give individual thread more space to use, which results in higher BTB hit rates. But BTB hit rate can also be degraded for multithread interference. We simulate the BTB behavior of GShare branch predictor with configuration of total BTB buffer ranged from 512 to 32K. During simulation, perfect branch direction prediction is assumed. Figure below shows the BTB mishit rates for 2 and 4-way multithreaded uniprocessor for TPC-W shopping workload.

We can make sure from the data that when size of BTB is enlarged, the BTB his rates increase as well. Size benefit of shared BTB offsets the conflict interference of multithread, therefore shared BTB performs
better than private BTB. Another reason we provide is that although inter-thread conflicts exist, each entry of BTB buffer is taken advantage differently among threads, and that situation equals that one thread can make use of some entries of other threads. The third reason is a fairly interesting one. Although very few, there are entries with same target address among different threads. We suspect this is caused by benchmark pattern, different threads of web interaction with shopping workload can execute same program or code segment sometime.

Figure 5: multithreading effect on BTB of GShare predictor

We can state from results above that shared BTB can achieve relatively good BTB hit rates. But it still needs more work to improve it for BTB hit rate is important for instruction prefetch and recovery work. Slightly better BTB hit rate can enhance system performance greatly. The BTB mishit is mainly caused by conflicts inside one thread and among threads. To address this problem, some algorithm similar to victim cache is presented to minimize BTB conflicts. We name the algorithm victim BTB in the same fashion of victim cache and explain in detail in following section.

4. Victim BTB

The basic idea of victim BTB is almost the same as that of victim cache. When one entry of BTB buffer is about to be replaced, its content is saved in some victim buffer for a short time. When that entry is taken for target prediction, both entries in entry and in victim buffer are inspected for correct target address. Inspecting victim buffer takes more time than inspecting BTB therefore there exits some algorithm for
victim buffer to exchange its content with BTB, keeping most matching target information in BTB. Inter-thread victim is different from intra-thread victim in inspecting and exchanging algorithm since they take different kind of conflict as research issue.

### 4.1 Inter-Victim design & result

Inter-thread aims at minimizing the loss caused by conflicts of multithread interference. Supposing thread 0 occupies the BTB entry and now thread 1 is running, when thread 1 takes that entry for target prediction, it will save its own target address in that entry, overwriting the content of thread A. When thread 0 is switched to run and takes that entry for target prediction, a BTB mishit is resulted. That BTB mishit can be saved if entry content of thread 0 can be buffered for a while. When thread 0 want to take that entry, the buffered target can be returned as correct target prediction.

Perfect inter-thread victim is provided to minimize the number of that kind of BTB mishit. Each thread in processor is assigned individually with a small size of buffer called inter-thread victim. And each entry in BTB is assigned to an owner thread if that thread updates its content. In implementation, that means a data field of thread id, two bits in our simulation environment, is added into entry data structure. If one BTB entry owned by some thread, assumed thread A, is about to be overwritten by other thread B, content of that entry is written back to inter-thread victim of thread 0 before thread 1 updates BTB entry. When thread 0 is switched to run and takes that entry for target prediction, content in victim buffer is considered as predicted target. Also system writes content of that entry to victim buffer of owner thread, updates entry with content in inter-thread victim of thread A, sets thread 0 as owner thread of that entry. Therefore when thread 0 uses that entry again later before it is switched out, it is not necessary to inspect victim buffer any more for entry match. FIFO is taken as victim entry replacement policy. Figure shows design of perfect inter-thread victim.

In the subfigure 1, the entry is occupied by thread 0 with target address A, thread 1 is running and wants to update it. So entry owned by thread 0 is evicted to victim buffer of thread 0, and the owner of that entry in BTB is changed to thread 1, which is shown in subfigure 2. In subfigure 3, thread 0 is switched to run and asks for target address of that entry. Since it finds out that the owner of that entry is thread 1, not itself, it inspects its own victim buffer for matched evicted entry. In subfigure 4, if matched entry is found in
victim buffer, it is restored back to BTB, and the original entry owned by thread 1 is evicted to victim buffer of thread 1, also the owner of that entry in BTB is changed to thread 0.

Figure 6: design of perfect inter-thread victim BTB

However, there is inherent defect in perfect inter-thread victim design. If size of inter-thread victim is more than a couple of items, inspecting the inter-thread victim buffer for entry match is expensive in hardware and execution time, similar as matching a fully-associative cache. The disadvantage of inspecting can offset the benefit of time saving of inter-thread victim, for computing out the target from instruction usually takes several cycles. Therefore perfect inter-thread victim needs some improvement to work realistically. We call the improved algorithm as semi-inter victim.

The idea of semi-inter victim is that no inspection of victim buffer is needed, instead contents of victim buffer is restored back to BTB during thread-context switches. If restored entry is not made use during one thread switch, it will be discarded when conflicting with another thread. Otherwise that useless entry will occupy the precious space of victim buffer, degrading target prediction performance. Figure shows design of semi-inter thread victim.

Figure 7: design of semi-inter thread victim BTB

Subfigure 1 and 2 are identical to those of perfect inter-thread victim design. In the subfigure 1, the entry is occupied by thread 0 with target address A, thread 1 is running and wants to update it. So entry
owned by thread 0 is evicted to victim buffer of thread 0, and the owner of that entry in BTB is changed to thread 1, which is shown in subfigure 2. Subfigure 3 shows what happens during thread context switch. Thread 0 restores all evicted entries in its victim buffer to BTB, and evicts all corresponding entries in BTB to victim buffer of their owner threads. Then it changes owner of all related entries to itself. Therefore when thread 0 wants to use those entries for target address prediction, it will only inspect entries in BTB since those entries belong to it. Searching in victim buffer for matched item is not needed any more.

Experimental result of inter-thread victim algorithm is shown in figure below. We can see that inter-thread victim algorithm is slightly effective in improving the total target prediction precision. Considering multithread interference only account for a small fraction of total BTB prediction, this result is acceptable. And since there are more multithread interferences in BTB of small size, inter-thread victim algorithm is more effective. In fact, when BTB is large enough, inter-thread algorithm almost has no effect at all.

The result of semi-inter thread victim is slightly worse than perfect algorithm, but it is still fairly better than no victim design. Since it is more realistic, we promote semi-inter thread victim as appropriate algorithm for minimizing effects of inter-thread conflicts. The perfect algorithm can be considered as upper-limit of semi-inter algorithm.

![Figure 8: effectiveness of perfect-inter and semi-inter thread victim BTB](image-url)
4.2 Intra-Victim design & result

Intra-thread victim aims at minimizing the loss caused by conflicts inside individual thread. This kind of conflicts is caused by size limit of BTB buffer. Different PC addresses can index same BTB entry; therefore interleaved appearance of multiple PC addresses, all indexing same BTB entry, can severely degrade BTB prediction performance by overwriting content of that entry frequently. This kind of conflicts counts for about 3/4 of total mishits with our workload. An almost replicated victim buffer algorithm, intra-thread victim, is provided. Design of intra-thread victim is quite straightforward.

![Figure 9: effectiveness of intra-thread victim BTB](image)

Experimental data below shows similar performance improvement as victim cache. Although intra-victim only contains a couple of items, it is fairly effective in reducing intra-thread conflicts. From data we can see that an intra-victim with size of 2 can effectively minimize the BTB mishit rate. This is because intra-thread interference account for a large part of BTB mishits.

4.3 Final results

The inter-thread victim and intra-thread victim algorithm can be integrated together to minimize both the inter-thread conflicts and intra-thread conflicts. Since these two kinds of conflicts are main cause of BTB prediction performance degradation, the final victim BTB algorithm can improve BTB hit rate greatly. Figure below shows some experimental results.

We can see from data that victim algorithm is effective when BTB size is small. It is because large size can resolve interference greatly. When BTB size is larger than some threshold such as 8K, the victim
algorithm almost exerts no influence on overall branch prediction performance. But that kind of configuration can approach nearly 100% BTB hit rates and little space is left for improvement.

![Graph showing branch prediction performance](image)

Figure 10: effectiveness of both inter & intra-thread victim BTB

## 5 Related Work

Various multithreading techniques are used to hide latency, but little research is done on its interaction with branch prediction. The work in this paper is an extension and more than an extension to the work in [3]. [3] simulates coarse-grain multithreading’s effect on BHR and BHT. But due to some implementation mistakes, the results for private BHR with shared BHT are not correct. Private BHR with shared BHT is actually a very good scheme to improve prediction accuracy. No other branch prediction structures are studied, and no other optimization strategies are exploited.

APRIL[1] is a coarse-grain multithreaded processor to be used in a cache-coherent multiprocessor called ALEWIFE. It achieves high single-thread performance as well as high processor utilization, by executing instructions from a given thread until an exception condition like a synchronization fault or remote memory operation occurs. There is no mention of private data structures used for branch prediction, so we assume all are shared. The branch prediction accuracy would probably be improved if some structures such as BHR and RAS could be private to each thread, but the benefit could be small if an exception condition doesn’t happen very frequently so it would be a waste of hardware to keep branch prediction context for a thread which won’t be swapped back in for quite a while.
[8] develops an analytical model of multithreaded processor behavior taken into consideration cache degradation due to multithreading. [10] studies the effect of the size of a multithread working set have on cache conflict misses, and when multiple hardware contexts could be most effective. Neither discusses the interaction of branch prediction accuracy and multithreaded processor behavior, especially the impact of multithreading to different branch prediction optimization strategies.

[7] proposes the interleaved multiple-context processor, essentially a fine-grained multithreading processor, with a small number of contexts. They also share branch prediction data structures. If branch pattern history is employed in branch prediction, private BHR should be quite beneficial since thread switching are quite often to tolerate short latency. There is quite some work on fine-grain multithreading [2,9] but all have not addressed the topics covered in this paper.


[11] proposes a technique called simultaneous multithreading (SMT) which permits multiple independent threads to issue multiple instructions each cycle. It achieves significant throughput improvements because it exploits the advantage of favoring for fetch and issues those threads most efficiently using the processor each cycle. It claims that simultaneous multithreading has a dual effect on branch prediction -- while it puts more pressure on the branch prediction hardware, it is more tolerant of branch mispredictions. In this architecture, most branch prediction structures are shared but each hardware context has a private RAS. It recognizes that better branch prediction will be beneficial but it is not further explored in the paper.

So we believe the work in this paper is a good starting point in the research of the interactions between multithreading and branch prediction. Better branch prediction accuracy will enhance system performance of multithreading architecture, while different multithreading -- coarse-grained, fine-grained, simultaneous, or 2-thread, 4-thread, 8-thread, dynamic -- will make varying optimization strategies effective. There is much more work to be done before we can draw a clean picture of the relationship of the two.
6. Conclusions

In this paper, effect of multithreading processors on branch prediction is discussed and data structures of three branch predictors, GShare, Bimodel, McFarling, are examined and optimized to achieve better branch performance. Data structures and implementation of typical two-level adaptive branch predictor are discussed in detail. From experimental data of TPC-W shopping workload, we state that in multithreading processors, BHR and RAS should be kept private to threads so as to minimize the multithreading interference while BHT and BTB should be shared among multiple threads in order to utilize the size benefit of sharing space.

Particularly, BTB performance is optimized by introducing victim buffer to minimize influence of inter thread and intra thread conflicts, which we call victim BTB algorithm. Specifically inter-thread and intra-thread victim BTB algorithms are designed and implemented. Experimental results show that victim BTB algorithm performs effectively when BTB size is not too large.

References


