Improving Conditional Branch Prediction on Speculative Multithreading Architectures

Chitaka Iwama, Niko Demus Barli, Shuichi Sakai, and Hidehiko Tanaka

Graduate School of Information Science and Technology, The University of Tokyo,
7-3-1 Hongo Bunkyo-ku, Tokyo 113-8654, Japan
chitaka@mtl.t.u-tokyo.ac.jp

Abstract. Dynamic conditional branch prediction is an indispensable technique for increasing performance in modern processors. However, currently proposed schemes suffer from loss of accuracy when applied to speculative multithreading CMP architectures. In this paper, we quantitatively investigate this problem and present a hardware scheme to improve the prediction accuracy. Evaluation results show that an improvement of 1.4% in average can be achieved in SPECint95.

1 Introduction

Speculative Multithreading has been proposed in some Chip Multiprocessors (CMPs) [6–9] to accelerate performance when executing single thread programs. A speculative multithreading CMP partitions a program into threads and speculatively execute them in parallel across its multiple processing units (PUs). In such execution, however, a number of problems emerged. One of them is the loss of accuracy of conditional branch prediction. Currently proposed conditional branch predictors rely heavily on branch correlations to achieve high accuracy. However, in speculative multithreaded execution, information on these correlations may not be available.

In this paper, we quantitatively investigate this problem and propose a hardware scheme to improve the prediction accuracy. The main idea of this scheme is to exploit the locality of branch correlations within a static thread, and to use a centralized history table to manage branch information from the distributed PUs.

2 Conditional Branch Prediction on Speculative Multithreaded Execution

In this section, we investigate the accuracy of currently available conditional branch predictors when applied to speculative multithreaded execution. We choose five representative predictors for the investigation: bimodal predictor, global predictor, gshare predictor, per-address predictor, and a hybrid predictor of global and per-address predictor.

Bimodal predictor[1] is the simplest among the five predictors. It consists of a table of saturating counters indexed by the program counter. Global predictor, gshare predictor, and per-address predictor are variants of Two-level Predictors[2], which use two levels of branch history. The hybrid predictor[3, 4] is a combination of global and per-address predictor, currently adopted in Alpha 21264 processor[5]

For the investigation, we simulated a 4-PU CMP, with a 4-kbyte conditional branch predictor in each PU. The PU is implemented as a 6-stage out-of-order superscalar processor with 64-entry instruction window, four functional units, two load-store units, and a 2k-entry speculative store buffer. We assumed perfect caches and all accesses to memory are completed in two cycles.

We defined a *thread* as a connected subgraph of a control flow graph with exactly one entry point. Inter-thread control and data dependencies are allowed. A program is statically partitioned into threads by a compiler. Innermost loop iterations are first marked as threads. For the rest parts of the control flow graph, the biggest possible subgraphs that meet the thread model's requirements are selected as threads. During the execution, threads are dynamically predicted and scheduled into the PUs. Inter-thread register dependencies are synchronized whereas inter-thread memory dependencies are handled speculatively. If a memory dependence violation is detected, the violating thread and all its successors will be squashed and restarted.

For clarity, we make distinction between static and dynamic threads. *Static thread* is used to refer to a portion of control flow graph identified as thread at compile time, whereas *dynamic thread* is used to refer to a stream of instructions of a static thread actually executed. Unless stated to be *static*, the word *thread* will be used to indicate a *dynamic thread*.

We compared the prediction accuracy in speculative multithreaded execution, with the accuracy in single threaded execution in a single PU. Eight programs from SPECint95 were used for the evaluations. The parameters were adjusted so that there are 10 to 25 millions of conditional branch instructions executed for each simulation run. In the following results, predictions made within misspeculated threads are not accounted.

Figure 1 shows the average prediction accuracy. Except for the bimodal predictor, the results showed a significant loss of prediction accuracy in speculative multithreaded execution. The accuracy degradation of the bimodal predictor is largely due to the increased time to train counter. But other predictors suffered mainly from the unavailability of correct branch information. There are basically two reasons why this information is not readily available.

1. A PU does not have access to the results of branches executed in other PUs. Not only the distributed nature of CMP architecture prevents such accesses, but also the speculative multithreaded execution does not guarantee that the results of branches from predecessor threads are available when a thread is started.
2. Since consecutive threads are scheduled to different PUs, a PU has to execute non-successive threads in sequence. The results of branches from these non-

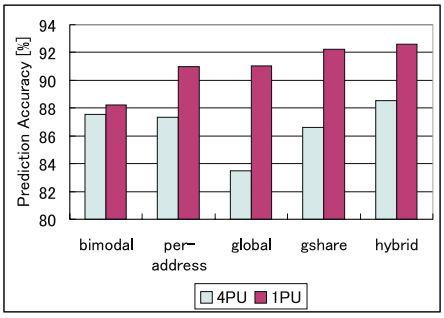

Fig. 1. Speculative multithreading effect on branch prediction accuracy.

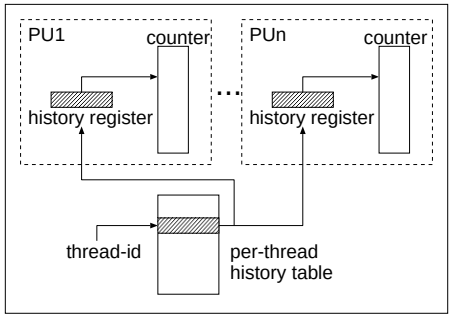

Fig. 2. Per-thread branch predictor.

successive threads, however, are recorded successively, resulting in a broken history information.

3 Improving Prediction using Per-thread Branch Predictor

To improve the branch prediction on speculative multithreading architectures, information on branch correlations should be used in more effective way. We propose a prediction scheme we call *per-thread branch prediction*. The main idea behind this scheme is to exploit the locality of branch correlations within a static thread.

Figure 2 illustrates the structure of per-thread branch predictor. Each PU has a prediction unit identical to a global predictor. In addition, there is a globally visible *per-thread history table* indexed by static thread identifiers. Each entry of this table records history of branches that belong to the same static thread.

When a PU starts executing a thread, it retrieves the thread's history from the table and initializes its history register with that value. During execution, branches are predicted in the same way as the global prediction scheme. After the execution of the thread is finished, the current value of the history register is written back to the per-thread history table. In case a thread needs to be restarted due to a misspeculation, the corresponding PU reinitializes its history register with the latest history from the per-thread history table. In this way, contamination of history register by misspeculated threads can be reduced.

Per-thread history table tries to maintain correct correlation information of branches that belong to the same static thread. However, when a PU retrieves a history of a thread from the table while a preceding copy of the same thread is still in execution anywhere else, the table cannot provide the PU with the most up-to-date history. Rather, it can only provide history from the last committed thread.

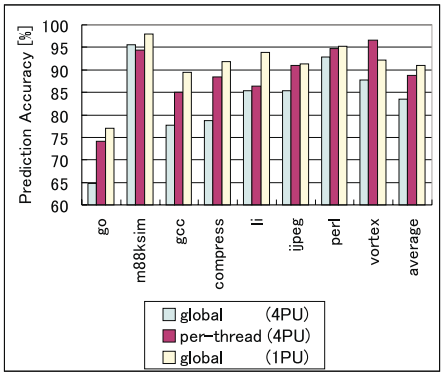

Fig. 3. Prediction accuracy of per-thread predictor and global predictor.

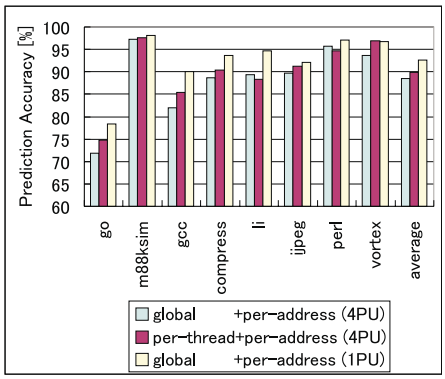

Fig. 4. Prediction accuracy of (per-thread + per-address) hybrid predictor and (global + per-address) hybrid predictor.

Another possible drawback of the per-thread prediction scheme is that it requires accesses to a centralized table before and after executing a thread. However, since these accesses can be carried out in parallel with other thread initialization/retirement processing, it is unlikely to cause a bottleneck problem to the architecture's critical path.

4 Evaluations

We first compared the performance of the per-thread predictor with that of a global predictor. Evaluation environment is identical to the one previously described in section 2. We assumed that the per-thread predictor consists of four 4-kbyte global predictors, one in each PU, and an additional 3.5-kbyte per-thread history table (2k-entry $\times$ 14 bits). Per-thread history table is a direct-mapped table indexed by the lower address bits of the first instruction in the thread. Table access delay is assumed to be two cycles.

Figure 3 shows the prediction accuracy of the global predictor and the per-thread predictor in speculative multithreaded execution. Prediction accuracy of the global predictor in single threaded execution is also shown as reference. It can be observed that the per-thread predictor improved the prediction accuracy of the global predictor on almost all applications, 5.3% in average.

We then combined the per-thread predictor with a per-address predictor to form a new hybrid predictor, and compared its accuracy with the hybrid predictor of global and per-address predictor. The new hybrid predictor is assumed to have four 4-kbyte local predictor structures, one in each PU, and a 3-kbyte per-thread history table (2k-entry $\times$ 12 bits). Figure 4 shows the simulation results. Except for *li* and *perl*, prediction accuracy was improved for all applications, by 1.4% in average.

One reason why some applications did not benefit from the per-thread prediction scheme is that, due to the parallel execution of threads,the per-thread history table was not always able to provide a correct per-thread history . Another possible reason is that the scheme only exploits branch correlation within a static thread. Thus, it was not effective for programs that make extensive use of global variables. Branches in these programs have a tendency to be predictable only when global correlation information is used.

5 Concluding Remarks

This paper investigated accuracy loss problem of conditional branch prediction on speculative multithreading CMP architectures and proposed per-thread prediction scheme to overcome the problem. Per-thread prediction uses a centralized history table to manage history of branches that belong to the same static thread. By exploiting the locality of branch correlations within a static thread, our scheme helped to improve the prediction accuracy by 1.4% in average.

The influence of intra-thread branch prediction to the overall performance largely depends on the architecture design. For example, in architectures that have PUs with deep pipeline and operate on threads whose size are relatively large, intra-thread branch prediction should be an important factor for their performance. We plan to make further investigation and perform more detail evaluations on this issue.

References

1. J. Smith: A Study of Branch Prediction Strategies. *Proc. 8th Annual Int'l Symp. Computer Architecture,* 135–148, 1981.
2. T. Yeh, Y. Patt: A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. *Proc. 20th Annual Int'l Symp. Computer Architecture,* 257–266, 1993.
3. S. McFarling: Combining Branch Predictors. *Technical Report TN-36, Digital Western Research Laboratory,* 1993.
4. P. Chang, E. Hao, T. Yeh, Y. Patt: Branch Classification: A New Mechanism for Improving Branch Predictor Performance. *Proc. 27th Annual Int'l Symp. Microarchitecture,* 22–31, 1994.
5. R. Kessler: The Alpha 21264 Microprocessor. *IEEE Micro, March-April,* 24–36, 1999.
6. K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, K. Chang: The Case for a Single Chip Multiprocessor. *Proc. 7th Int'l Symp. Architectural Support for Programming Languages and Operating Systems,* 2–11, 1996.
7. L. Hammond, M. Willey, K. Olukotun: Data Speculation Support for a Chip Multiprocessor. *Proc. 8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 48–69, 1998.
8. G.S. Sohi, S. Breach, T.N. Vijaykumar: Multiscalar Processors. *Proc. 22nd Annual Int'l Symp. Computer Architecture,* 414–425, 1995.
9. V. Krishnan, J. Torellas: A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers, Vol. 48, No. 9.* 866–880, 1999.