Improving Conditional Branch Prediction on Speculative Multithreading Architectures

Chitaka Iwama,[†] Niko Demus Barli,[†] Shuichi Sakai[†] and Hidehiko Tanaka[†]

Dynamic branch prediction is an indispensable technique for increasing performance in modern processors, and has been actively investigated in the last decade. However, those methods are not readily applicable to speculative multithreading architectures. In this paper, we discuss problems that emerge when conventional conditional branch predictions are used on speculative multithreading architectures, and propose a prediction scheme to improve the prediction accuracy. Evaluation results show that by using our scheme, the average of prediction accuracy of SPEC95int applications can be improved from 89.7% to 90.4%, approaching the accuracy on single threaded execution of 92.9%.

1. Introduction

With the current trend of microprocessor architecture toward deeper pipeline and wider issue width, branch prediction has become a critical part of microprocessor design. Branch prediction addresses two basic problems: predicting direction of conditional branches and predicting address from which the next instructions will be fetched. In this paper, we will concentrate on the former one, and discuss conditional branch predictions in the context of speculative multithreading architectures.

Many sophisticated conditional branch prediction schemes have been proposed, including two-level adaptive predictors [2, 3], gshare predictor [4], and hybrid predictors [4, 5]. These predictors and their variants exploit past history of the predicted branch, history of recent branches, and other branch correlations to accurately predict conditional branches [6, 7]. However, while these predictors can achieve high accuracy in current superscalar processors, the situation is different when they are applied to speculative multithreading architectures.

Speculative multithreading has been proposed in some Chip Multiprocessor (CMP) architectures [9–14], to accelerate CMP performance when executing single-thread programs. These architectures partition a program into threads and speculatively execute them in parallel. These threads are not necessarily independent, and may include some sort of dependences (control, register, memory or combination of them). The processors aggressively execute these speculative threads, and take necessary actions when the speculative execution failed.

Due to distributed nature of CMP's processing units (PUs) and unordered occurrences of consecutive branches during speculative threads execution, conventional conditional branch prediction schemes will not perform as well as in the case of single threaded execution. This paper discusses this problem in more details, and propose a hardware scheme to improve the performance of conditional branch prediction on speculative multithreading architectures.

The following sections are organized as follows. Section 2 explains the speculative multithreading model assumed in this paper and our simulation environment. Section 3 shows and discusses performance degradation of conventional conditional branch prediction when used on speculative multithreading architectures. Section 4 presents our proposal of branch prediction scheme and explains the rationale behind it. Section 5 reports evaluation results of the proposal. Finally, section 6 concludes the paper.

2. Execution Model and Simulation Environment

2.1 Execution Model

This paper assumes a speculative multithreading model similar to Multiscalar architecture [11]. As illustrated in figure 1, the architecture has 4 PUs and a centralized thread control unit, whose job is to schedule threads into PUs. A

† Graduate School of Information Science and Technology, The University of Tokyo

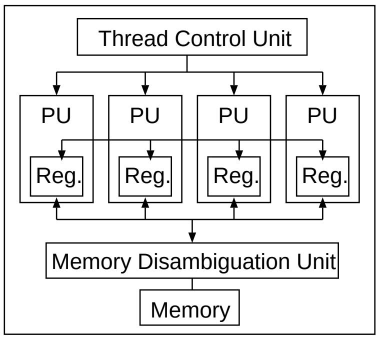

Fig. 1 CMP model.

thread in this model can comprise a basic block, multiple basic blocks, loop iteration, entire loop or any group of basic block that has exactly one entry point. In this paper, we partition a program into threads using a method proposed in [15]. First, thread candidates are identified from program's control flow graph using structural analysis. Then, thread candidates that comprise of outermost loop iteration are identified and marked as threads. Thread boundaries are also put at function calls so that a new thread will be spawned whenever a function is called. For other parts of control flow graph, the biggest possible thread candidates are identified and marked as threads.

Both control and data dependences between threads are allowed. Control dependences will be administered and controlled by the thread control unit. Data dependences will be handled as follows:

- Register dependences are synchronized. An instruction that uses a register value produced by a preceding thread is not allowed to execute before that value is made available by the producer.
- Memory dependences are handled speculatively. Threads will execute LOADs speculatively with the expectation that its predecessors will not STORE a value into the same location at later time. Dedicated hardware is responsible to check this collision. In case a collision occurs, violating threads are squashed and restarted.

2.2 Simulation Environment

We used eight applications from SPEC95int suite as benchmarks. First we analyzed source programs and partitioned them into threads. Thread boundaries are indicated by a special instruction implemented for this purpose. After building the binaries, we took execution traces using an instruction level simulator. Execution parameters for each program are adjusted so that the program will finish the execution in 180-370 million instructions.

The traces were then simulated using a speculative multithreading simulator. Each PU is implemented as an out-of-order superscalar processor. A 16-entry speculative store buffer is used to support speculative multithreading. Register communication latency between a producer thread and a consumer thread is assumed to be 1 cycle. It is also assumed that a register value can be sent to another thread as soon as the last instruction which writes into the register retires. In case a memory data dependence violation is detected, the execution of the violating thread and all of its successors will be restarted after 1 cycle of restart delay. Currently, we assume a perfect cache for the simulation. All accesses to memory are assumed to finish in 2 cycles. We also assume a perfect next-thread prediction by the thread control unit.

3. Conditional Branch Prediction and Speculative Multithreading

3.1 Characterization of Conditional Branch Predictions

Among many available conditional branch prediction schemes, we limited our investigation to four representative prediction schemes: bimodal predictor, global predictor, per-address predictor, and hybrid predictor such as the one found in Alpha 21264 processor. Figure 2 illustrates the structures of these predictors.

Bimodal predictor [1] is the most simple predictor among the four predictors. It uses a table of saturating counters, usually 2 or 3 bits long, indexed by the low-order bits of the branch address. The appropriate counter is incremented for each taken branch and decremented for each non-taken branch. Thus bimodal predictor distinguishes repeatedly taken branches from repeatedly not-taken branches. Repeatedly taken branches will be predicted to be taken, and repeatedly not-taken branches will be predicted to be not-taken.

To achieve a higher prediction accuracy, predictors that use two levels of branch history have been proposed [2, 3]. Global predictor and per-address predictor are the two representative versions of them.

Table 1 Branch prediction unit parameters.

predictor	size	history length	history table ent.	counter length	counter table ent.
bimodal	4KB	-	-	2 bits	16K
global	4KB	14 bits	1	2 bits	16K
per-address	3.3KB	11 bits	2K	2 bits	2K
hybrid	3.6KB				
global	1KB	12 bits	1	2 bits	4K
per-address	1.6KB	10 bits	1K	3 bits	1K
selector	1KB	-	-	2 bits	4K

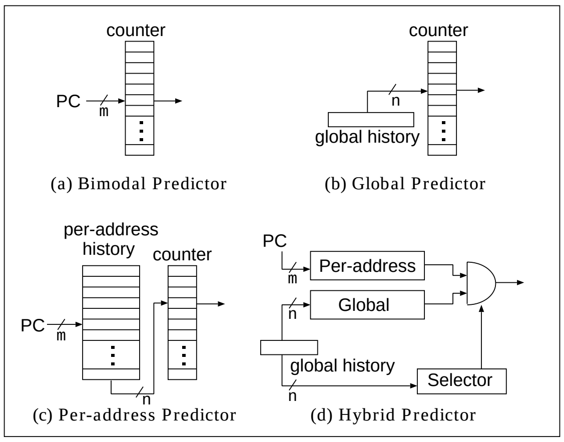

Fig. 2 Conditional branch predictors.

Global predictor is designed to predict current branch direction using behaviors of recent branches. It typically consists of a global history register and a table of saturating counters, as illustrated in figure 2(b). The history register records directions taken by the most recent branches, and is used to index the table of counters. For a given global history pattern, an appropriate counter is incremented or decremented, and is used to predict branch direction in the same way as the bimodal scheme. Since in most cases the current branch has some kind of relation with recent branches, the use of global history register enables the global predictor to be more accurate than the bimodal predictor.

Per-address predictor exploits repetitive patterns of branches. The most common example of this behavior is loop control branches. Instead of a single global history register, per-address predictor maintains histories of each branch independently. As shown in figure 2(c), there are typically two tables used in the predictor. The first one is a history table indexed by the low-order bits of the branch address. Each entry records the most recent branch directions of the branch mapped to it. The second one is a table of saturating counters, indexed by the per-address history selected from the first table.

Evaluation results show that global and per-address predictor have a comparable accuracy in respect of each other [4]. Since each predictor uses different approaches for predicting branches, there are classes of branches that can be predicted better with one predictor, while other classes of branches can be predicted with more accuracy using the other predictor. This leads to the idea of combining the advantages of both predictors into one prediction scheme.

Hybrid predictor shown in figure 2(d) is a combination of global and per-address predictor, and is currently adopted in Alpha 21264 processor [8]. This predictor comprises a global predictor, a per-address predictor, and a selector. The selector is a table of saturating counters, indexed by a global history register. This register is the same register used to index the global predictor. The counters in the selector hold information on whether the global or the per-address is more accurate for a given recent branches history. Using this information, the selector associates a currently predicted branch to either global or per-address predictor which is most likely to be more accurate. Using similar amount of hardware resources, this hybrid approach is proved to be more accurate than the stand-alone predictors [4–6].

3.2 Prediction Accuracy on Speculative Multithreading Architecture

In section 1, it has been briefly discussed that currently available branch prediction schemes will suffer when applied to Chip Multiprocessor which supports speculative multithreading. To verify this, we compared branch prediction accuracy on 4 PU speculative multithreading architecture described in section 2.1 with that on conventional single threaded execution architecture. We assumed that each PU has an independent conditional branch prediction unit. We simulated eight applications from SPEC95int and observed prediction accuracy for four branch prediction schemes described in the previous subsection. Table 1 shows parameters of branch prediction schemes used in this experiment.

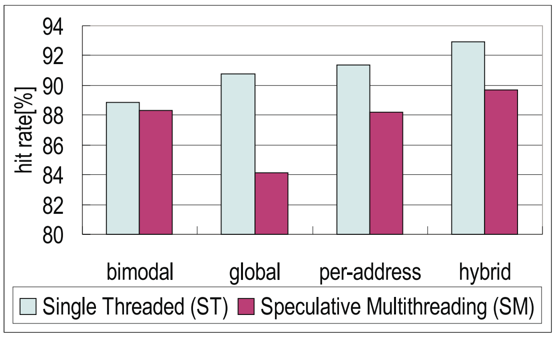

Fig. 3 Speculative multithreading effect on branch prediction accuracy.

Figure 3 shows the average of prediction hit rate. Performance loss is observed for all predictors, although the amounts vary among prediction schemes. Global predictor suffers the most, losing its accuracy for 6.6%. Per-address and hybrid predictor suffer moderate performance degradation of 3.2%, while the accuracy loss of bimodal predictor is only 0.5%. Overall, our result shows that when used in the speculative multithreading architecture, global predictor and per-address predictor are less accurate than a bimodal predictor, and a hybrid predictor can only maintain a narrow performance margin to a bimodal predictor.

Analyzing the characteristics of speculative multithreading execution, we can identify some reasons for this branch prediction accuracy degradation.

(1) *Increasing time to train counters.* This is the only factor that contributes to the performance loss of the bimodal predictor. Since this loss is almost negligible compared to those of the other predictors, we can deduce that the increased training time is not the limiting factor for performance in global, per-address or hybrid predictor.

(2) *Increasing time to record repetitive patterns.* For example, to record a repetitive pattern of a loop control branch, it takes 4 times more number of iterations for speculative multithreading 4PUs than for 1PU carrying out single threaded execution. This affects prediction accuracy at the beginning of execution and increases compulsory misprediction. But, again, it does not contribute much to the overall accuracy degradation.

(3) *Incomplete branch history.* The branch history in each predictor does not contain the results of branches executed in other PUs. This prevents a predictor to exploit correct correlation information and leads to poor performance, as shown by per-address, global and hybrid predictor.

(4) *Global history inaccessibility.* Since we assume a PU does not have access to global history register of the other PUs, a global predictor cannot use recent branches' history from its directly preceding thread. It can only use history of branches from previous thread executed in the same PU. This explains why a global predictor suffers more severely than a per-address predictor.

The next section will describe how we can deal with this accuracy problem by using a scheme we call per-thread branch predictor.

4. Improving Prediction using Per-thread Branch Predictor

4.1 Per-thread Branch Predictor Idea

Our experimental result has shown that the global predictor performs the worst on speculative multithreading architectures. However, if we can provide the predictor with sufficient information on branch correlations, it has potential to perform much better. By modifying the global predictor and improving its accuracy, it is also expected that the hybrid predictor will benefit from it and achieve a better overall performance.

As we have already discussed, there are two main factors for the poor performance of the global predictor: incomplete branch history and inaccessibility of the global history. While the latter problem can be solved if we can maintain a globally accessible history register, the former problem is much more difficult to confront. Since consecutive threads are executed in parallel on different PUs, there are both spatial and timing problem to build a complete branch history. Especially the timing problem prevents us to use the most recent branches' directions at the beginning of each thread execution, i.e.

when a thread begins to execute, the history of branches from a preceding thread is not available unless all the branches in the thread have executed.

The difficulty to build a complete branch history in a global predictor originates from the characteristic that the global predictor tries to maintain a history of branches within a global scope, i.e. the whole program. This leads us to the idea of narrowing the scope of the history to a single thread rather than the whole program. Threads in speculative multithreading are usually created at loop boundaries and function call boundaries, so that we can expect enough correlation locality to predict a branch correctly. During the execution of a thread, the history of branch directions are recorded in a register local to the PU, and is used to predict branches in an identical way to the global prediction. When the execution is finished, we save this history to a table accessible by all PUs. This saved history will be used to initialize the history register when the same thread is executed again in the future.

Using the above scheme, we still have to face the timing problem, because when a PU starts to execute a thread and tries to retrieve initial history value, the preceding copy of the same thread may still be in execution on the other PU. We solve this problem by letting the PU retrieve the history value from the last committed thread. Although we cannot use the most up-to-date version of the thread's history, we could expect that most threads would behave in the same way most of the time. Thus, there should be little performance loss.

Since the proposed prediction scheme exploits the locality of branch correlation inside a thread, we call this scheme *per-thread branch prediction.*

4.2 Implementation of Per-thread Branch Predictor

Figure 4 illustrates the structure of a per-thread branch predictor. Each PU has a predictor unit whose structure is identical to that of a global predictor. In addition, there is a globally visible per-thread history table, indexed by thread identifier. Each entry of this table records branch histories of the thread associated with it. When a PU starts executing a thread, it will retrieve branch history from the table, and initialize its history register with the retrieved history. During the execution of the thread, predictions are carried out locally in the PU in the same way as the global prediction scheme. After the execution is finished, current value of the PU's history register is written back to the per-thread history table.

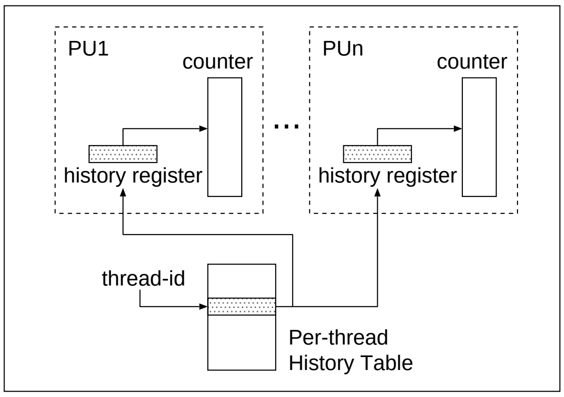

Fig. 4 Per-thread branch predictor.

In case a memory violation occurs and a thread needs to be restarted, the PU will reinitialized its history register with the value currently written in the per-thread history table. In this way, the history register is not contaminated by misspeculated executions.

Per-thread branch predictor has some beneficial features as follows:

(1) *Simple to implement.* It only needs an extra table to hold per-thread branch histories. This table can be implemented as a direct mapped table to reduce the amount of hardware required. It will also be shown later that this table only needs a size of about 3 KBytes to achieve an optimum performance.

(2) *No additional delay added to branch predictor's critical path.* Although a PU needs to retrieve history from the per-thread history table at the beginning of each thread execution, it predicts branches locally in a manner identical to a global predictor. Thus, our per-thread prediction scheme should have similar critical path as a global predictor and should be feasible enough to implement.

5. Evaluations

In this section, we first evaluate prediction accuracy of a per-thread branch predictor and compare it with that of a global predictor. Then, we combine a per-thread predictor and a per-address predictor to form a hybrid predictor and compare its performance with the original hybrid predictor of global and per-address predictor. Finally, we investigate how the prediction accuracy changes when the number of entries in the per-thread history table is varied, and discuss some implementation aspects for the per-thread history table. The detail of the simulation approach is described in section 2.

5.1 Comparison of Per-thread Predictor with Global Predictor

We investigated the performance of a per-thread predictor and a global predictor on speculative multithreading architectures. In both cases, each PU has a structure identical to the 4KByte global predictor previously described in table 1. Per-thread predictor uses an additional 3.5KByte per-thread history table (14 bit history × 2048 entries). The table is indexed by low-order address bits of the first instruction in the thread.

Figure 5 shows the simulation results for eight applications from SPEC95int benchmark suite. It also shows prediction accuracy of the global predictor in single threaded execution for comparison. The per-thread predictor significantly improved the performance of the global predictor, 4.9% in average. Only *m88ksim* suffered a slight performance degradation. It is because, in *m88ksim*, most of the threads were too small for the per-thread predictor to exploit enough branch correlations. But in most cases, the per-thread predictor could effectively exploit locality of branch correlations within threads. The average hit rate is 89.0%. Comparing with the result shown in figure 3, it can also be seen that this hit rate is higher than that of the bimodal and the per-address predictor when used in speculative multithreading execution.

These results shows that the per-thread predictor makes effective use of branch correlations within threads and improves prediction accuracy of the global predictor.

5.2 Combining Per-thread Predictor with Per-address Predictor

In the previous evaluation, the per-thread predictor did not perform better than the hybrid of global and per-address predictor. However, we can expect a better accuracy if we combine the per-thread predictor with a per-address predictor and form a new hybrid predictor.

To verify this, we compared the accuracy of the new hybrid predictor with that of the original hybrid predictor. We assumed that each PU has a 4KByte global and per-address hybrid predictor shown in table 1, and added a 3KByte per-thread history table (12 bit history × 2048 entries) to use it as a per-thread and per-address hybrid predictor.

The simulation results are shown in figure 6. The prediction accuracy of the global and per-address hybrid predictor in single threaded execution is also shown for comparison. As we expected, the hybrid of per-thread and per-address predictor was more effective than a per-thread predictor of the same size. It outperformed the original hybrid of global and per-address predictor for many applications. By using the per-thread prediction scheme, the prediction accuracy for *go*, *gcc*, *ijpeg*, and *vortex* was improved by 1.7%-2.9%. The improvement covers more than 30% of the performance loss caused by speculative multithreading. In average, the hybrid of per-thread and per-address predictor was the best performing predictor for the speculative multithreading architecture, outperforming the original hybrid predictor by 0.7%.

However, the new hybrid predictor was less accurate than the original hybrid predictor for *m88ksim* and *li*. This can be explained by the fact that the characteristic of the per-thread predictor is closer to a per-address predictor when the sizes of threads are small. Compared with the other applications, *m88ksim* and *li* comprise of small threads so that little benefit was gained by combining a per-thread predictor with a per-address predictor.

Figure 7 shows how often an element predictor is selected inside the hybrid predictors. Global predictor is selected 37.8% of the times in average when the programs are executed in single threaded environment, but this number is down to 26.9% in speculative multithreading execution. This proves that the performance degradation of the hybrid predictor is caused mainly by the poor accuracy of the global predictor. But when the global predictor is replaced with a per-thread predictor, the per-thread predictor is selected for as much as 54.9% of the branch executions, because it is more accurate than a global predictor.

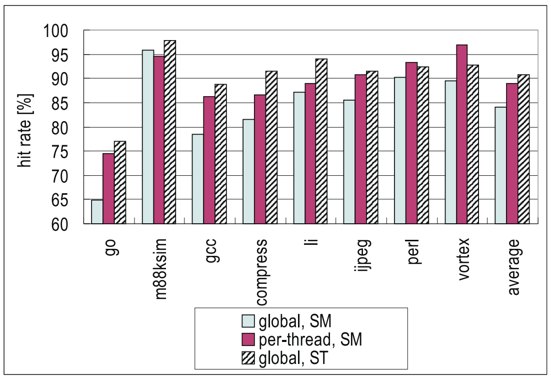

Fig. 5 Hit rate of per-thread and global predictor.

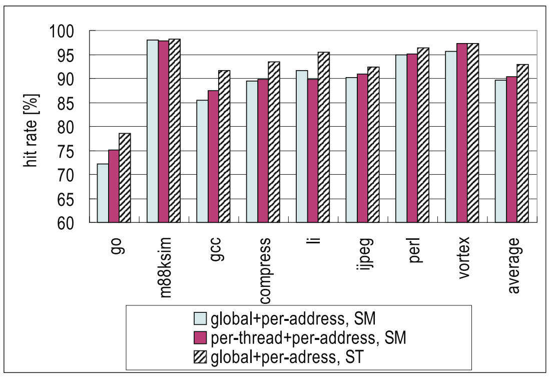

Fig. 6 Hit rate of (per-thread+per-address) hybrid predictor and (global+per-address) hybrid predictor.

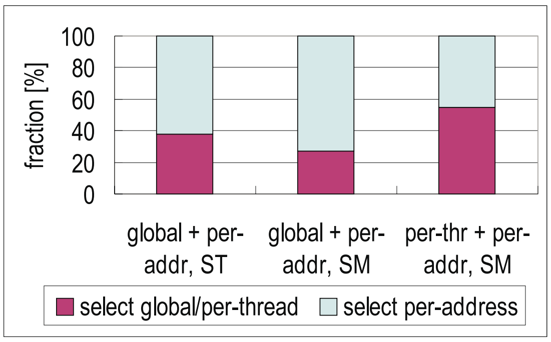

Fig. 7 Fraction of no. of predictions made by a particular predictor inside a hybrid predictor.

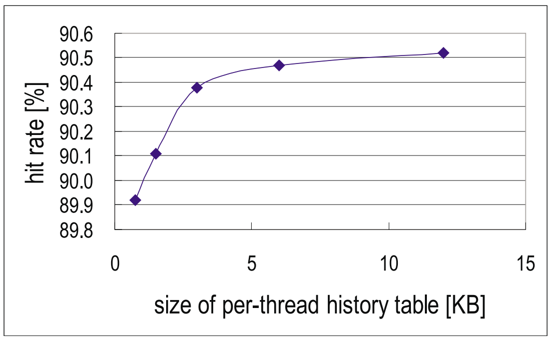

Fig. 8 Prediction accuracy with varying per-thread history table size.

5.3 Varying the Size of Per-thread History Table

Figure 8 shows how the accuracy of the per-thread and per-address hybrid predictor changed when the size of the per-thread history table was varied. We fixed the history register length to 12 bits, and varied the number of entries in the per-thread history table. All other parameters are the same with those we used in the previous section.

According to figure 8, a table size of 3 KBytes, which has 2048 entries of 12 bit history, will give an optimum accuracy while keeping the hardware small. It should be noted, however, that the SPEC95int benchmark we used in this evaluation consists of relatively small applications compared to current real world applications. Bigger programs will be partitioned into larger number of threads, and tend to use more entries in the per-thread history table. But since there is principle of time locality that should also apply to threads, i.e. a thread that is being executed now is likely to be executed again in the near future, we could expect that the need for the per-thread table size will grow in slower pace compared to the growth rate of real world applications sizes. Moreover, in case a big and fast table is required, we can adopt certain implementation techniques available, such as the ones used to implement hierarchical caches in modern processors.

6. Concluding Remarks

This paper first pointed out and verified that the performance of conventional conditional branch predictors is degraded when used on speculative multithreading architectures. We investigated and classified problems for four types of branch prediction schemes: bimodal predictor, global predictor, per-address predictor, and hybrid of global and per-address predictor.

Having analyzed the problems, we proposed a branch prediction scheme to improve prediction accuracy in speculative multithreading execution. This prediction scheme is based on a global predictor, but exploits locality of branch correlations within a thread rather than the whole program. We called the scheme per-thread prediction. A per-thread predictor consists of local prediction units within each PU that are similar to global predictors, and a globally accessible per-thread history table. Each entry of this table holds history for a thread associated with it. This history is used to initialize the history register in a PU before the execution of the thread begins, and is updated when the execution is finished.

The evaluation results showed that the per-thread predictor is more accurate than a global predictor by 4.9%, and when combined with a per-address predictor, outperforms the hybrid of global and per-address predictor by 0.7%. We also argued that our per-thread predictor is simple and cost effective, and should be feasible enough to implement.

We believe that the per-thread prediction approach has a good potential to improve the branch prediction on speculative multithreading architectures. However, further investigations are needed to evaluate how its prediction accuracy depends on the thread partitioning algorithm, and how it contributes to the overall performance of the processor. We also plan to characterize the performance degradation of conventional branch predictors in more detail and refine the prediction scheme.

References

[1] James E. Smith, *A Study of Branch Prediction Strategies*, Proceedings of the 8th Annual International Symposium on Computer Architecture, pp. 135-148, 1981

[2] Tse-Yu Yeh and Yale N. Patt, *Alternative implementations of two-level adaptive branch prediction*, Proceedings of the 19th Annual International Symposium on Computer Architecture, pp. 124-134, 1992

[3] Tse-Yu Yeh and Yale N. Patt, *A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History*, Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 257-266, 1993

[4] Scott McFarling, *Combining Branch Predictors*, Technical Report TN-36, Digital Western Research Laboratory, 1993

[5] Po-Yung Chang, Eric Hao, Tse-Yu Yeh and Yale N. Patt, *Branch Classification: A New Mechanism for Improving Branch Predictor Performance*, Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 22-31, 1994

[6] Marius Evers, Sanjay J. Patel, Robert S. Chappell and Yale N. Patt, *An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work*, Proceedings of the 25th Annual International symposium on Computer Architecture, pp. 52-61, 1998

[7] Cliff Young, Nicolas Gloy, and Michael D. Smith, *A Comparative Analysis of Schemes for Correlated Branch Prediction*, Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 276-286, 1995

[8] Richard E. Kessler, *The Alpha 21264 Microprocessor*, IEEE Micro, pp. 24-36, March-April 1999

[9] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson and Kunyung Chang, The Case for a Single Chip Multiprocessor, Proceedings of the 7th International Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), pp. 2-11, 1996

[10] Lance Hammond, Mark Willey, and Kunle Olukotun, *Data Speculation Support for a Chip Multiprocessor*, Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 48-69, 1998

[11] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar, *Multiscalar Processors*, Proceedings of the 22nd Annual International Symposium on Computer Architecture, 1995

[12] Venkata Krishnan, Josep Torellas, *A Chip-Multiprocessor Architecture with Speculative Multithreading*, IEEE Transactions on Computers, Vol. 48, No. 9, September 1999

[13] 小林 良太郎, 岩田 充晃, 安藤 秀樹, 島田 俊夫, 非数値計算プログラムのスレッド間命令レベル並列を利用するプロセッサ・アーキテクチャ*SKY*, 並列処理シンポジウム JSPP'98, pp.87-94, Jun 1998

[14] 鳥居 淳, 近藤 真己, 本村 真人, 西 直樹, 小長谷明彦, *On Chip Multiprocessor* 指向 制御並列アーキテクチャ*MUSCAT*の提案, 並列処理シンポジウム JSPP'97, pp.229-236, 1997

[15] Niko D. Barli, Hiroshi Mine, Shuichi Sakai, and Hidehiko Tanaka, *A Thread Partitioning Algorithm using Structural Analysis*, 情報処理学会 計算機アーキテクチャ研究会, ARC-2000-139 Vol. 2000, No. 24, pp. 37-42, Aug 2000