# Compiler-Assisted Thread Level Control Speculation

Hideyuki Miura, Luong Dinh Hung, Chitaka Iwama, Daisuke Tashiro, 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
{hide-m,hung,chitaka,tashiro,niko,sakai,tanaka}@mtl.t.u-tokyo.ac.jp

**Abstract.** This paper proposes two compiler-assisted techniques to improve thread level control speculation in speculative multithreading execution. The first technique is to identify threads which have exactly one successor and the successor's address is statically known (we call these threads *fixed-successor threads*), and use a small full associative buffer to predict the successors. This technique reduces aliasing in the original thread predictor and increases the overall prediction accuracy. The second technique is to insert validation information at points where the address of the successor thread is resolved. This *early validation* technique enables the processor to validate thread prediction earlier and reduce the penalty when a misprediction occurs. Our evaluation results show that for a 2K-entry predictor, a 5.8% average performance improvement can be achieved by combining the two techniques.

## 1 Introduction

Recently, numerous researches have focused on speculative multithreading to exploit thread level parallelism (TLP) from sequential applications [1, 3, 5, 6, 8, 9]. Speculative multithreading partitions sequential programs into threads and executes them in parallel. In contrast to conventional parallelization approach, data or control dependences may exist among threads and various speculation techniques are performed to extract more parallelism.

This paper proposes two compiler-assisted techniques to improve the efficiency of thread level control speculation in speculative multithreading architectures. Our techniques aim to both increase the accuracy of thread prediction and reduce the penalty when a misprediction occurs. The first technique is to identify threads which have exactly one successor and the successor's address is statically known. We call these threads *fixed-successor threads*. We use a small full associative buffer we call *fixed-successor cache* to predict their successors. Using this technique we can reduce aliasing in thread predictor and improve the overall prediction accuracy.

The second technique is to insert validation information at points where the address of the successor thread is resolved. Using reachability analysis, the compiler identifies control independent points of successor threads and inserts *assert* instructions. When a processing unit encounters an *assert* instruction, it notifies the thread control unit of the address of successor thread. This *early validation* technique enables thread prediction to be validated even before the execution of the current thread finishes, thus reducing the penalty when a misprediction occurs. Our evaluation results show that combining the above two techniques improve the achieved performance by 5.8% in average.

The remaining parts of this paper is organized as follows. Section 2 summarizes related work on thread level control speculation. Section 3 describes baseline architecture used in this research. The compiler-assisted techniques we propose are described in section 4. Section 5 shows and discusses the evaluation results. Finally, section 6 concludes the paper.

## 2 Related Work

Approaches taken by speculative multithreading architectures to perform control speculation can be broadly classified into two categories. The first category is to support a limited control speculation, in which threads are spawn only at highly predictable points such as loop iterations and function boundaries. Many of the previously proposed architectures fall into this category [1, 3, 5, 6, 9]. Although this approach minimizes penalty caused by control misspeculation, it limits the potential to exploit higher parallelism especially in integer applications. In these applications, innermost loop iterations, for example, typically only occupy a small portion of the total executed instructions [6, 10].

The second category of control speculation is to aggressively predict and spawn threads as early and as many as possible. This approach is used in Multiscalar [4, 8] and also employed in this research. In Multiscalar, the compiler inserts statically known addresses of thread candidates to thread header. This technique reduces the hardware cost of the predictor. However, this results in longer prediction latency since the address must always be brought from the instruction cache. Furthermore, the size of thread header relative to the thread size is substantially large. This overhead may result in instruction cache contamination and a reduced overall throughput. Our approach of using fixed-successor characteristics is a compromise of this trade-off. We suppress the overhead of thread header while still contributing to the increase in prediction accuracy.

The approach of early thread validation is useful to reduce the effective misspeculation penalty. From another perspective, it can also be seen as a technique to exploit control independence characteristics of instructions. Control independence has been shown to be an important source of parallelism and its implementation issues are under investigation [2, 7]. Early validation enables our architecture to exploit parallelism from control independent instructions in successor threads of the current point of execution.

## 3 Baseline Architecture

Throughout the paper, we assume a speculative multithreading chip multiprocessor as shown in figure 1. It comprises of a thread control unit (TCU) and four processing units (PUs). Each PU has a private register file and L1 caches. Finally, an L2 unified cache is shared by all the PUs.

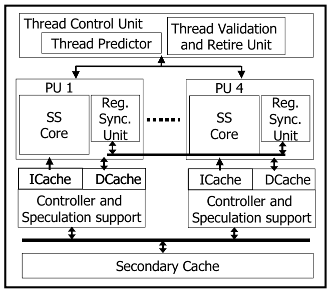

**Fig. 1.** Baseline architecture.

A sequential program is partitioned into threads statically by a compiler. A thread is defined as a connected subgraph of a static control flow graph with a single entry node. It may comprise a basic block, multiple basic blocks, loop body, or entire function. Most threads consist of 10-20 dynamic instructions. During program execution, the processor's TCU predicts threads following the program order, and schedule each thread to a processing unit in a round-robin fashion. We assume a register communication mechanism to synchronized inter-thread register dependencies. We also assume a hardware support for thread level memory speculation support similar to [5].

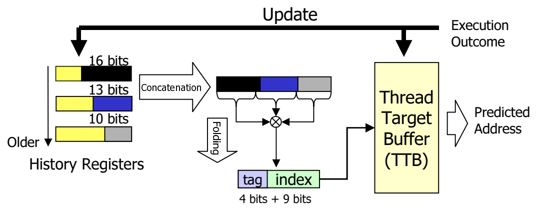

**Fig. 2.** Baseline path-based predictor. The TTB is a 2K-entry 4-way set associative table.

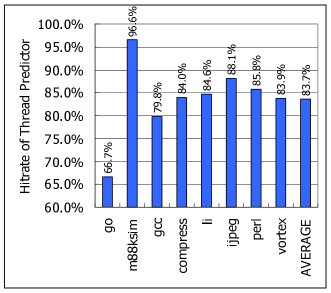

**Fig. 3.** Accuracy of thread prediction using the baseline predictor.

TCU includes a thread predictor which predicts address of thread that follows the current thread in program order. In this paper, we use a 2K-entry path-based predictor as a baseline predictor. Figure 2 illustrates the structure of the predictor. It consists of a set of history registers, which record addresses of the three most recently executed threads, and a Thread Target Buffer (TTB) which holds the addresses of successor threads. We used a 4-way set associative table for the TTB with four-bit tags to reduce the effect of conflict aliasing.

The index for accessing the TTB is generated by applying a hashing function on history registers. Here we adopt a hashing function proposed by Multiscalar [4]. In this scheme, lower address bits of the history registers are concatenated to form an intermediate history. The final history is then constructed by folding the intermediate history. Folding is done by dividing the intermediate history into several, identical length subfields and then XORing those subfields. Four bits of the final history are used as tag and the rest of the bits are used as an index for TTB.

Figure 3 shows the accuracy achieved using this thread predictor. While a high accuracy is achieved for *m88ksim*, the accuracy of *go* and *gcc* is low. For these programs, in addition to a low predictability characteristic, the aliasing effect caused by the index folding further reduced the prediction accuracy achieved. Our technique of using *fixed-successor* information as will be described in the next section, helps to reduce this aliasing by avoiding highly predictable threads to occupy entry in the TTB.

## 4 Compiler-Assisted Thread Level Control Speculation

In this section, we explain the two compiler-assisted techniques we propose to improve thread level control speculation in speculative multithreading architectures.

### 4.1 Fixed-Successor Threads

We defined a *fixed-successor thread* as a thread which has exactly one successor and the successor's address can be determined statically. Fixed-successor threads occupy 30% of the total number of threads executed in SPEC95int benchmark. Although successors of fixed-successor threads are theoretically highly predictable, due to aliasing effect, the thread predictor fails to predict them. For example, in *go*, the misprediction rate for successors of fixed-successor threads is 15.8%. Furthermore, the successors unnecessarily occupy entries in TTB increasing the effect of aliasing.

We use the characteristics of fixed-successor threads to reduce such mispredictions. A small full associative buffer is added to exclusively predict the successors of fixed-successor threads. We call the buffer *fixed-successor cache.* The organization of the predictor is shown in figure 4. When predicting a successor thread, both of the TTB and fixed-successor cache is looked up. If a valid entry is found in the fixed-successor cache, the predicted address obtained from it is used. Otherwise, the predicted address obtained from the TTB is used. TTB is always updated when a misprediction occurs, while fixed-successor cache is only updated when the mispredicted thread is a successor of a fixed-successor thread.

To identify fixed-successor threads during the execution, the compiler put marks on their thread headers. Fixed-successor threads can be statically determined using thread reachability analysis on the program control flow graph. The compiler also inserts starting address of the successor into the header. When a fixed-successor thread is assigned to a PU, the PU decodes the header and notifies the thread control unit of the successor address. When the TCU detects

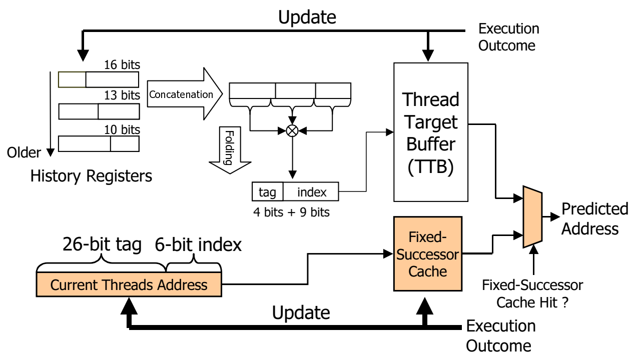

**Fig. 4.** Structure of thread predictor with a 64-entry fixed-successor cache.

a misprediction, it flushes the current thread and restarts execution of a new thread using the address supplied by the PU. We call this technique *early fixed-successor validation*. This technique can also be interpreted as a special case of *early validation* described below.

## 4.2 Early Validation

In a conventional approach, the validation of thread prediction must wait until execution of the current thread finishes. The content of program counter (PC) is then notified to thread control unit and used to validate the prediction. When a misprediction occurs, the previously predicted thread is flushed and a new successor thread is restarted using the notified address.

Misprediction penalty of the conventional approach is unnecessarily large. Thread control speculation can be validated earlier since the possible paths taken by the control flow converge as the execution advances. The misprediction penalty can be reduced if the thread control unit knows some information about the possible path a thread may take.

We defined *assert point* and *reject point* to exploit this characteristic. Assert point is a point where the successor thread can be uniquely determined. In contrast, reject point is a point where a successor thread becomes unreachable. To identify these points, the compiler performs reachability analysis of the control flow for each statically defined thread in the program.

Figure 5 illustrates how the compiler identifies *assert* and *reject* points. Suppose there is a thread which comprises four basic blocks (BB1∼BB4) as shown in figure 5(a). The thread has three successor thread candidates: Thread A, Thread B, and Thread C respectively. Using control flow reachability analysis, the compiler identifies assert points where only one of the three threads is reachable. Similarly, the compiler also identifies reject points where one or more of the successor threads become unreachable points. Figure 5(b) shows the position of assert and reject points.

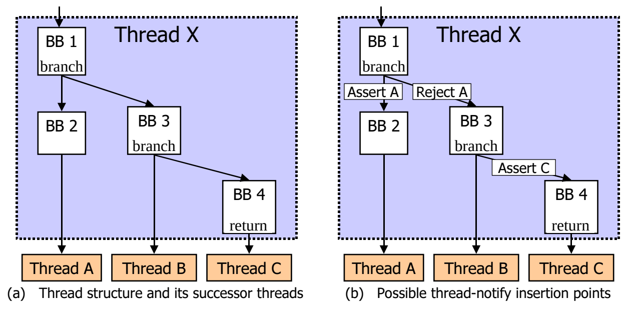

**Fig. 5.** Identification of assert and reject points in a thread.

For each assert or reject point, the compiler inserts a special instruction to notify the TCU about the reachability of the successor thread candidates. We call the instructions *assert instruction* and *reject instruction* respectively. The instructions carry a pointer to a register that will hold the starting address of a successor thread in consideration. Another instruction to set the thread address into the register is also inserted. This results in two additional instructions for each assertion or rejection of a thread.

The instructions are processed as follows. When a PU executing a thread encounters an assert instruction, it notifies and sends the address of the successor thread to TCU. TCU then checks if the address of thread predicted earlier matches the address notified by the processing unit. If a mismatch occurs, the TCU knows it has mispredicted the successor thread and can start the procedure to flush the mispredicted thread and restart the execution of a correct successor thread.

Similar notification mechanism is also used for reject instructions. When the TCU is notified by a PU, it does the address matching process. In case of notification initiated by a reject instructions, misprediction can be detected if the notified address matches the predicted address. If it is the case, the thread control can flush the mispredicted thread, and optionally repredict and restart an execution of another thread.

## 5 Evaluations

### 5.1 Simulation Environment

To evaluate the performance advantages of using our compiler-assisted techniques, we conducted simulations using a trace-based speculative multithreading CMP simulator. The simulator simulates a 4-PU CMP as previously described in section 3. Each PU is a 4-issue 10-stage out-of-order superscalar core with a 32KB L1 data and instruction cache (2-cycle latency). L2 cache shared by all PUs is assumed to have an infinite size (always hit, 6-cycle latency). Eight

SPEC95int benchmark applications are used for the simulations. The input parameters are adjusted so that the execution finishes between 100-300 million instructions.

## 5.2 Simulation Results

We simulated the following five control speculation models.

- **TTB**: This is our baseline model. Threads are predicted using a path-based predictor with a 2K-entry TTB as described in figure 2.
- **TTB+FSC**: This model integrates a 64-entry fixed-successor cache into the TTB model. The structure of the predictor is shown in figure 4.
- **TTB+FSC+EFSV**: In addition to TTB+FSC model, this model employs early fixed-successor validation. Header of a fixed-successor thread holds the address of its successor. After the header is decoded, the address is notified to the thread control unit for validation.
- **TTB+FSC+EFSV+ASSERT**: In addition to TTB+FSC+EFSV, this model further employs early validation technique using assert instructions as previously described in section 4.2.
- **TTB+FSC+EFSV+ASSERT+REJECT**: In this model, we further added early validation technique using reject instructions. Since reject instruction by itself does not tell the correct successor address, the thread control unit has to repredict a new thread when there is a misprediction. Here, since our first interest is on the potential of early validation using reject instructions, we assume that the prediction always succeeds.

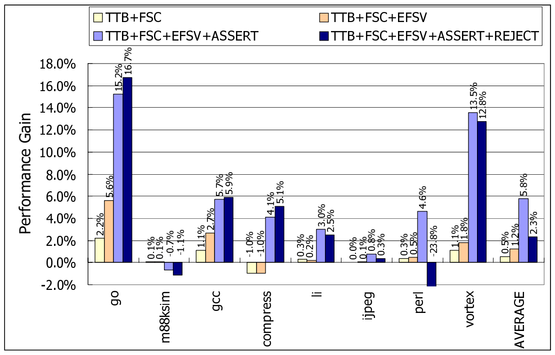

**Fig. 6.** Performance gain relative to TTB model.

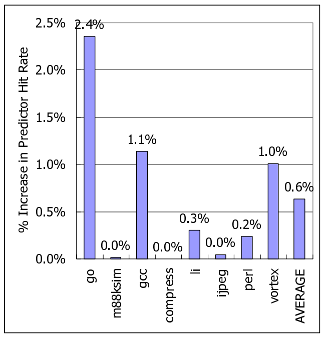

**Fig. 7.** Prediction accuracy improvement using fixed-successor cache.

Figure 6 shows the performance gain of the last four models over the baseline TTB model. TTB+FSC model improved the performance by 0.5% in average.

The performance gain is especially noticeable for *go*, *gcc* and *vortex*, which have large footprint characteristics. For these programs the improvements are 2.2%, 1.1%, and 1.1% respectively. These gains came from a better prediction accuracy achieved by integrating fixed-successor cache into the thread predictor. The increase of prediction accuracy for each application is shown in figure 7.

Employing early fixed-successor validation as in TTB+FSC+EFSV model further increased the performance gain relative to the baseline model to 1.2% in average. Similar to TTB+FSC model, the improvements are especially observable in *go*, *gcc*, and *vortex*. In these applications, early fixed-successor validation salvages the performance loss when a successor of a fixed-successor thread is mispredicted by both the TTB and the fixed-successor cache.

Applying early validation technique using assert instruction as in TTB+FSC+EFSV+ASSERT model, significantly increased the performance to an average of 5.8% over the baseline model. Especially in *go* and *vortex* the gains are substantially large: 15.2% and 13.5% respectively. This is because in *go*, the control flow is hard to predict and accuracy of baseline predictor is low. Therefore, early validation significantly reduced the penalty suffered from thread mispredictions. As for *vortex*, the largest part of mispredictions happens when predicting a successor thread of a thread exiting from a return instruction. In this case, since the return address is also notified using assert instruction, the misprediction penalty is significantly reduced.

Although the impact early validation technique using assert information is significant, further incorporating early validation using reject instructions is not beneficial for some applications. This is mainly because the execution overhead of inserted reject instructions is larger than the effect of reduced misspeculation penalty. Figure 8 shows the number of assert and reject instructions inserted as a ratio to the number of useful instructions. It can be seen that in *perl*, for example, the overhead of reject instructions was so large that even when we assume a perfect reprediction mechanism, the performance is degraded.

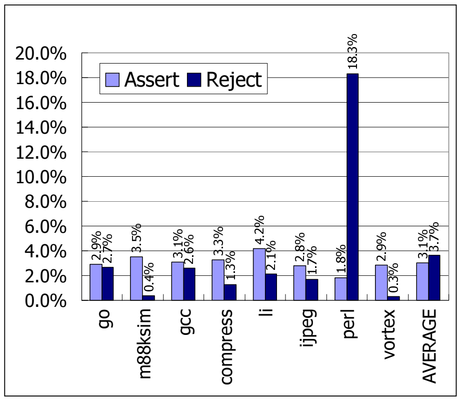

**Fig. 8.** Ratio of assert and reject instructions to the number of useful instructions.

Figure 9 shows the performance gain when we varied number of TTB entries. TTB+FSC+EFSV+ASSERT model achieved higher performance than TTB model by 8.6% at 1K-entry, and 2.8% at 16K-entry. The gain is smaller for a larger TTB, because TTB alone achieves higher prediction accuracy. However, using compiler assistance, TTB size can be reduced substantially while keeping

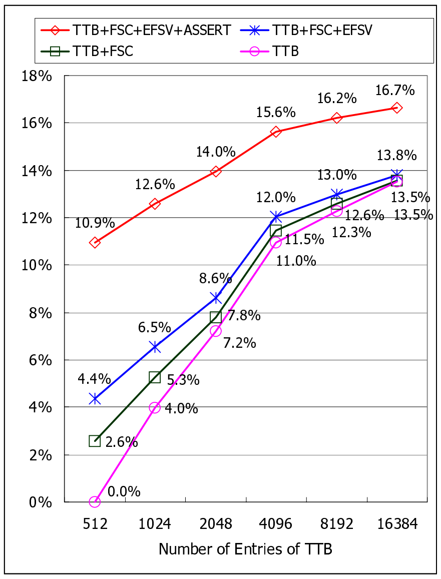

**Fig. 9.** Performance gain relative to 512-entry TTB.

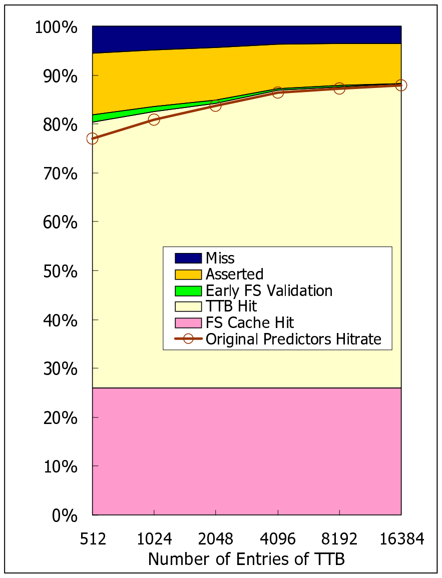

**Fig. 10.** Breakdown of thread predictions.

performance. For example, TTB+FSC+EFSV+ASSERT model with 1K-entry TTB shows better performance than 4K-entry TTB with no static information.

Figure 10 shows the breakdown of thread execution count when TTB+FSC+EFSV+ASSERT model is employed. It explains where the performance gain shown in figure 6 originated from. The summation of FS cache hit and TTB hit gives overall hit rate of thread prediction. The hit rate is higher than the original thread predictor used in the baseline model, which explains performance improvement made by fixed-successor information. The amount of asserted threads indicates the performance gained by lower missprediction penalty achieved with assert instructions. Those two factors together contribute to overall speed up.

## 6 Conclusion

This paper explored compiler-assisted techniques to improve thread level control speculation in speculative multithreading executions. The first technique used the compiler to identify *fixed-successor threads*, i.e. threads which have exactly one successor and the successor's address is statically known. Since successor of these threads are easy to predict, instead of the original path-based predictor, we used a small full associative buffer for the prediction. We call this buffer *fixed-successor cache*. This technique helps to reduce aliasing in the path-based predictor and increase the overall accuracy of thread prediction.

The second technique used the compiler to analyze the control flow structure of threads and insert validation information at points where the address of the successor thread is resolved. This information is used to validate thread prediction earlier, thus reducing the penalty when a misprediction occurs. From another perspective, this technique also enables the processor to exploit more parallelism from a successor thread when it becomes control independence relative to the current thread.

Our evaluation results showed that for a 2K-entry path-based predictor, a 5.8% average performance improvement can be achieved by combining the two techniques. The results also show that our techniques are especially useful when the aliasing effect in the original predictor is prohibitive.

## References

1. H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proc. of the 31st International Symposium on Microarchitecture*, 1998.
2. J. S. Eric Rotenberg, Quinn Jacobson. A Study of Control Independence in Superscalar Processors. In *Proc. of the 5th International Symposium on High-Performance Computer Architecture*, pages 115–124, 1999.
3. L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proc. of the 8th International Symposium on Architectural Support for Parallel Languages and Operating Systems*, pages 58–69, 1998.
4. Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control Flow Speculation in Multiscalar Processors . In *Proc. of the 3rd International Symposium on High-Performance Computer Architecture*, pages 218–229, 1997.
5. V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.
6. P. Marcuello, A. Gonzalez, and J. Tubella. Speculative Multithreaded Processors. In *Proc. of the 12th International Conference on Supercomputing*, pages 77–84, 1998.
7. E. Rotenberg, S. Bennett, and J. E. Smith. Skipper: A Microarchitecture for Exploiting Control-flow Independence. In *Proc. of the 34th International Symposium on Microarchitecture*, pages 4–15, 2001.
8. G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd International Symposium on Computer Architecture*, pages 414–425, 1995.
9. J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proc. of the 27th International Symposium on Computer Architecture*, pages 1–12, 2000.
10. J. Tubella and A. Gonzales. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *Proc. of the 4th International Symposium on High-Performance Computer Architecture*, pages 14–23, 1998.