計算機アーキテクチャ 149−12
（２００２．８．２２）

A Hardware/Software Approach for Thread Level Control Speculation

Luong Dinh Hung,† Hideyuki Miura,† Chitaka Iwama ,†
Daisuke Tashiro,† Niko Demus Barli,† Shuichi Sakai†
and Hidehiko Tanaka†

Speculative multithreading is a promising approach that exploits thread level parallelism from sequential programs. This paper focuses on thread level control speculation for a speculative multithreading architecture, and proposes a hardware/software techniques that improves its performance. Our approach is to use static analysis of control flow to validate dynamic prediction at an earlier stage, and also to reduce task of the dynamic thread predictor. We show that this approach reduces prediction miss penalty and, at the same time, improves the thread prediction accuracy.

1. Introduction

The limit of Instruction Level Parallelism (ILP) hinders superscalar microprocessors from achieving high performance against general purpose application [1]. The approach of exploiting Thread Level Parallelism (TLP) has proved to be very successful for several types of applications, such as multiprogrammed workloads or multithreaded applications. However, applying this approach to sequential programs still remains a challenge.

Recently, numerous researches have focused on speculative multithreadings to exploit TLP from sequential applications [2] [3] [7]. Those architectures break sequential programs into threads and execute them in parallel using multiple execution units. In order to ease the program partitioning, threads that are candidates for parallel execution are allowed to be dependent on each other, i.e. data dependences or control dependences may exist between threads. In exchange, additional hardware and software support is integrated to maintain sequential semantics.

Since the effectiveness of such parallelization is limited by the inter-thread dependences, speculative multithreading architectures perform various forms of thread level speculation. Threads are speculatively executed in parallel regardless of data or control dependences among them. Intensive researches have been done on the mechanism of thread level speculation. Data speculation can be performed using speculative load [7] or data value prediction [10], whereas control speculation is based on various form of control flow prediction.

This paper focuses on thread level control speculation to improve the performance of speculative multithreading execution. We address the speculation problem from two perspectives: improving the prediction accuracy and reducing the speculation miss penalty. We propose a combination of hardware and software approach: a thread prediction hardware assisted by static information inserted by a compiler. The static information is used to validate thread prediction at an earlier stage and also to reduce capacity and conflict misses of thread prediction table.

The remainder of this paper is organized as follows. Section 2 explains the thread level control speculation mechanism in our architecture model, and refer to how other proposed speculative multithreading models deal with inter-thread control dependences. Section 3 presents our proposed techniques. Section 4 evaluates the contribution of the techniques to the overall performance and discusses their trade-offs. Section 5 concludes the paper.

2. Thread Level Control Speculation

2.1 Baseline Architecture

Throughout the paper, we assume a speculative multithreading architecture described as follows. First, sequential programs are partitioned into threads at compile time. 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 20-30 dynamic instructions [8].

Those threads are then executed in parallel on a

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

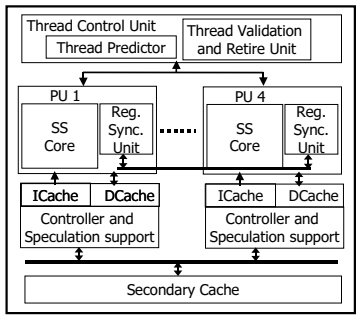


Fig. 1 Baseline architecture

chip multiprocessor that integrates four processing units on a single chip. Figure 1 illustrates the organization. Register files and first-level caches are distributed to each processing unit. For handling inter-thread data-dependences, we assume a register synchronization mechanism similar to the one found in [2], and a memory speculation mechanism similar to [7]. Inter-thread control dependences are handled by a centralized thread control unit. The unit includes a hardware we call *thread predictor*, which predicts address of the thread that should be executed next. The processing units are logically organized in the form of an one-directional ring. According to the suggestion made by the thread predictor, the thread control unit speculatively assigns a thread to the tail of the ring of the processing units. If the prediction fails, the thread control unit discards the misspeculated threads and recovers execution.

2.2 Importance of Thread Prediction

The thread prediction mechanism has a great impact on overall performance. Figure 2 shows an example of thread level control speculation. Let us consider the control flow graph shown in figure 2 (a), where thread B and C are control dependent on thread A. Which of the two threads should be executed cannot be determined until the last branch in thread A resolves. However, in order to gain performance, the thread predictor predicts which path will be taken so that the threads can be speculatively executed in parallel. In figure 2 (b), the predictor has successfully predicted thread B to be executed. The three threads are spawned successively after a short thread starting overhead, and completed within a minimum cycle time. On the other hand, figure 2 (c) shows a case in which the prediction turned out to be a miss. After the last branch in thread A resolved, the thread control unit is informed that thread 3 should be executed instead of thread B. Then the control unit flushes all the successors of thread B, and restart execution of thread C. It can be seen that a large number of cycles are waisted by the execution of misspeculated threads and the recovery of the execution states.

This example shows that the performance of the speculative multithreading execution is largely improved when the thread prediction hits, but suffers severely when the prediction misses. Thus, it is important to design a speculation mechanism that can maximize the thread prediction accuracy and/or minimize the speculation miss penalty.

2.3 Related Work

Various models of speculative multithreading have been previously proposed, each with different ways of handling control dependences.

Some architectures do not allow inter-thread control dependences. In SKY architecture, threads are only forked at control equivalent points [3]. This scheme avoids control misspeculation and eliminates its penalty. However, it sacrifices chances to achieve more performance gain in inherently parallel applications.

Multiscalar processor relies on thread level control speculation. A detail work has been done on its mechanism [6]. Multiscalar uses both hardware and software to achieve high thread prediction accuracy. Software assistance comes in the form of a thread header appended to each thread. Compiler includes in the header the number and type of the thread exit points, as well as addresses of the next threads if they are known at compile time. At runtime, a dedicated hardware predicts the control flow based on a similar scheme to conventional branch predictors. First, it predicts which exit of the current thread is most likely to be taken, using a Pattern History Table (PHT). Then, it predicts the address of the exit. The address is either obtained from the thread header or predicted using a Thread Target Buffer(TTB), which is an extension of a Branch Target Buffer. Although this approach is successful in achieving high prediction accuracy, it does not consider the possibility to achieve more performance by reducing prediction miss penalty.

3. Hardware/Software Approach

3.1 Concept

As already noted in the previous section, to im-

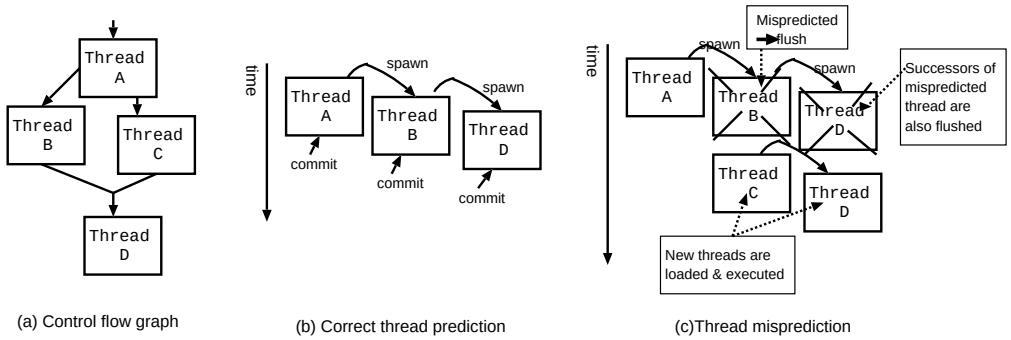


Fig. 2 Example of thread level control speculation

prove execution performance through thread level control speculation, it is important to achieve high prediction accuracy and low misspeculation penalty. Since thread prediction can be seen as an extension of branch prediction, it is possible to apply a number of sophisticated branch prediction mechanisms previously proposed. However, these mechanisms are mainly designed only to increase the accuracy of prediction. Alternatively, we investigate a method that use compiler assistance to reduce misspeculation penalty as well as to improve prediction accuracy.

Our approach is to let the compiler perform reachability analysis of the control flow graph and provide a mechanism to inform the thread control unit about thread reachability. Specifically, the compiler marks points in the control flow graph where a successor thread can be uniquely identified or where it becomes unreachable. Using this information, the thread control unit may be able to validate the prediction result at an earlier stage. The early validation permits an early recovery from the misspeculated execution states, leading to lower misprediction penalty.

It is also possible to use the thread reachability information to increase the accuracy of thread prediction. There are cases when a successor thread can be uniquely determined statically. We define a thread whose successor thread can be fixed before its execution begins as a *fixed-successor thread*. If the compiler marks the fixed-successor threads with appropriate information, the thread control unit is able to obtain the correct address of a successor thread without using the thread predictor, thus reducing the requirement for prediction table entries. It is expected that it will lead to a reduced capacity and conflict misses of prediction table and an increased overall prediction accuracy.

In the following subsections, the structure of thread prediction hardware used in this research is described. Then, the analysis method and mechanism for early control speculation validation is explained. Finally, the mechanism for handling fixed-successor threads is described.

3.2 Hardware Structure of Thread Predictor

In this research we use a thread predictor whose structure is shown in figure 3. Thread Target Buffer (TTB) is a table that caches the starting address of successor threads. The index for accessing the TTB is generated by applying a hashing function on history registers. History registers record starting addresses of the most recently executed threads. They are used to generate thread history pattern to exploit correlation among threads to achieve better prediction accuracy.

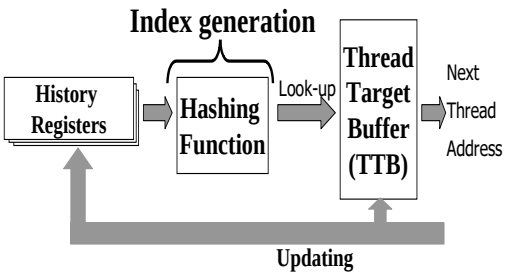


Fig. 3 Structure of thread predictor

To reduce conflict misses when accessing TTB, the hashing function must also effectively encode the thread history pattern into a designated length index. Here we adopt a hashing function proposed by Multiscalar group [6]. Figure 4 illustrates the index generation mechanism. First, lower address bits of the History Registers are concatenated to form an intermediate index. The

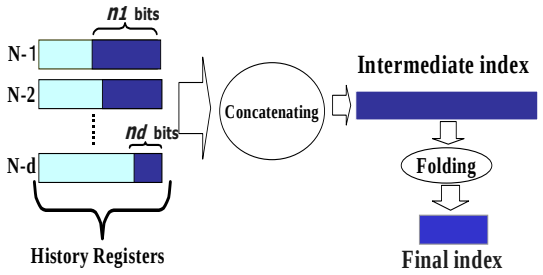
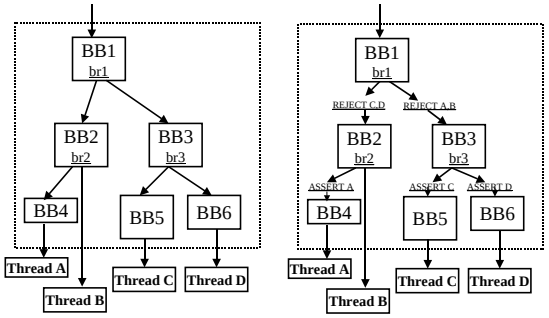


Fig. 4 Index generation mechanism



(a)Thread Structure (b) Insertion of assert & reject

Fig. 5 Illustration of sofware assisted early validation

final index is then constructed by *folding* the intermediate index. Folding is done by dividing the intermediate index into several, identical length subfields and then XORing those subfields.

3.3 Software Assisted Early Validation

This subsection explains how we implement the concept of early control speculation validation using compiler assistance. After partitioning a program into threads, the compiler does control flow reachability analysis for each thread. The analysis identifies points where a successor thread can be uniquely determined, or where it becomes unreachable. The compiler then inserted *assert* and *reject* instructions for the two cases respectively. The *assert* and *reject* instructions carry a pointer to a register that will hold starting address of a successor thread.

Figure 5 illustrates how the compiler inserted these instructions. Suppose there is a thread which comprises six basic blocks (BB1∼BB6) as shown in figure 5(a). It has four possible successors, thread A, thread B, thread C and thread D. Using control flow reachability analysis, the compiler identifies points where only one of the four threads is reachable and insert *assert* instructions. Similarly, the compiler finds points where one or more of the four threads become unreachable, and insert *reject* instructions. Figure 5 shows where those instructions will be inserted for the example.

Assert instruction indicates that the control flow will absolutely go to the designated successor thread. When a processing unit executing a thread encounters an *assert* instruction, it notifies the thread control unit and sends the address of the successor thread. Thread control unit then checks if the address of thread predicted earlier matches the address notified by the processing unit. If a mismatch occurs, the thread control unit 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.

In contrast to *assert* instruction, a *reject* instruction indicates that there is no possibility that the control flow will go to the designated successor thread. The notification mechanism is identical to the case of *assert* instruction. When the thread control unit is notified by a processing unit, it again does the address matching process. In case of notification initiated by *reject* instructions, misprediction can be detected if the notified address matches the 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.

3.4 Fixed-Successor Thread

As noted previously, we defined a fixed-successor thread as a thread that has only one successor whose address can be statically determined. Fixed-successor threads can be identified using the same reachability analysis explained in the previous subsection. When the compiler can assure that a thread may only have one successor, and the start address of the successor can be determined statically, it marks the thread as a fixed-successor thread. The compiler also inserts the starting address of the successor into the thread's header. The thread control unit will use this address to start the correct successor thread.

Figure 6 illustrates the flow of the thread prediction mechanism. First, the thread control unit checks if the thread is marked as a fixed-successor thread. In that case, the address of the successor thread is obtained from the thread header. Otherwise, it is predicted using the hardware predictor.

By exploiting the characteristics of fixed-

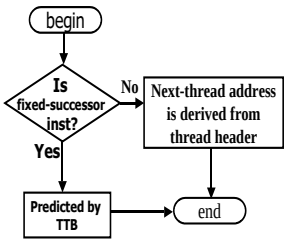


Fig. 6 Flow of thread prediction

successor threads, we can eliminate the possibility of mispredicting a successor of a fixed-successor threads. In addition, since the fixed-successor threads do not use entry in the TTB, it is also expected that the accuracy of the thread predictor can be further improved, resulting in more performance gain.

4. Evaluation

4.1 Simulation Environment

We use a trace-based simulator that models a 4-PU (processing unit) CMP described in section 2.1. The architecture configuration is summarized in table 1.

Parameter	Configuration
Pipeline stages	10
Arith/Logic units	4
Address units	2
Reorder buffer size	64
Load/Store queue size	20
Fetch/Decode/Rename/Retire width	4
L1 Instruction cache	32 kb - 2 cycles
L1 Data cache	32 kb - 2 cycles
L2 Unified cache	ideal - 6 cycles
Inter PU register comm.	1 cycle
TTB no. of entries	2048
TTB history depth	4
TTB index folding	3 times

Table 1 Configuration parameters

We prepared four types of binaries of SPECint95 benchmarks.

- **BASE**: no control flow information is added.
- **FS**: for each *fixed-successor* thread, its header is marked and equipped with the address of its successor thread.
- **FS+ASSERT**: in addition to *fixed-successor* information, *assert* instructions are inserted wherever possible.
- **FS+ASSERT+REJECT**: *reject* instructions are fully added.

The *fixed-successor* information and *assert* instructions are handled using the mechanism described in section3. As for *reject* instructions, in case a misprediction is detected due to a reject notification, the thread predictor repredicts and restarts a new successor thread. In order to evaluate the potential of using *reject* information, here we assume that the reprediction is always successful.

4.2 Results

Figure 7 shows the normalized execution time of SPECint95 programs. Binaries with *fixed-successor* thread information resulted in better performance for all programs, especially *go*, *gcc* and *vortex*. The average speed-up was 3.3%. Adding *assert* instructions gave further performance improvements in most cases. For *go* and *vortex*, the execution time was reduced by more than 8.5%. However, the effect of *assert* instructions varied among applications. For example, *ijpeg* and *m88ksim* required slightly longer execution time than in FS case. The *reject* instructions did not contribute to the performance. Except *compress* and *go* which showed limited improvement, all the programs suffered degraded performance. The execution time of *perl* was increased by more than 50%.

The positive effect of the additional control flow information comes from both early validation of the thread prediction and higher prediction accuracy. On the other hand, there is also a negative effect that comes from the increased number of executed instructions. Those factors are discussed in detail below.

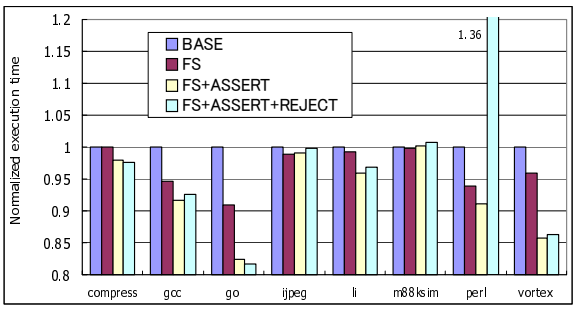


Fig. 7 Normalized execution time of eight SPECint95 programs

Prediction hit-rate. We found that, on average, 28.4% of executed threads are *fixed-successor* threads. Since the *fixed-successor* threads do not occupy entries in TTB, the thread predictor can achieve higher prediction accuracy. We verified

that average hit-rate of FS set was 2.8% higher compared to that of BASE binaries. The hit-rate improvement was considerable for *go*, *gcc* and *vortex*, whose control flow is relatively less predictable. This explains the outstanding performance improvement of those programs that we have observed in figure7.

Number of retired instructions. The number of retired instructions is shown in figure 8. Checking whether a thread is a *fixed-successor* thread is done by consulting the thread header.Therefore, BASE and FS binaries show the same number of retired instructions. Adding *assert* and *reject* information increased the number of instructions by 5.7% and 8.1%, respectively. In case of *assert* information, for most programs, the benefit of early misprediction validation was larger than the extra overheads due to the increased number of instructions. However, the reverse was true for *reject* information. Especially for *perl*, the number of instructions was increased by 44%, resulting in severe slowdown in performance as shown in figure 7.

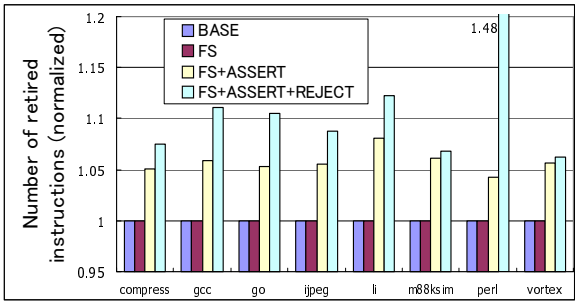


Fig. 8 Number of retired instructions (normalized)

5. Conclusion

This paper explored an approach of combining hardware and software to provide a better thread level control speculation. The approach is to let the compiler insert thread reachability information into the binaries. We introduced three types of information: *fixed-successor*, *assert* and *reject*. Using this information, the validation of the thread prediction can be initiated earlier, reducing the penalty when a misprediction occurs. Moreover, *fixed-successor* information is also useful for achieving higher prediction accuracy.

Our evaluation results showed that using *fixed-successor* information alone reduced the execution time by an average of 3.3%. Adding *assert* information reduced the execution time further by 3.7%. However, the gain achieved by inserting *reject* information was cancelled by the increased number of executed instructions.

More work is needed to improve the effectiveness of early misprediction validation. Currently, *assert* or *reject* instructions are inserted wherever possible. A smarter approach that inserts information only where a substantial gain is expected, might result in a better performance.

Acknowledgement

This research is partially supported by Semiconductor Technology Academic Research Center (STARC) Japan, under project name Research on Next Generation Low-power Chip Multiprocessors.

References

[1] David W. Wall, *Limits of Instruction Level Parallelism*, Proceedings of the 4th ASPLOS, pp.176-188, Apr. 1991

[2] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar, *Multiscalar Processors*, Proceedings of the 22nd ISCA, pp.414-425, Jun. 1995

[3] 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 JSPP98, pp.87-94, Jun. 1998

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

[5] Simonjit Dutta, Manoj Franklin, *Control Flow Prediction Schemes for Wide- Issue Superscalar Processor*, IEEE Transactions on Parallel and Distributed System, Vol 10, No.4, Apr. 1999

[6] Quinn Jacobson, Steve Bennett, Nikhil Sharma, J.E. Smith, *Control Flow Speculation on Multiscalar Processors*, Proceedings of the 3rd HPCA, pp.218-229, Feb. 1997

[7] Lance Hammond, Benedict A.Hubbert, Michael Siu, Manokar K.Prabhu, Michael Chen, Kunle Olukotun, *The Stanford Hydra CMP*, IEEE Micro, pp.71-84, Dec. 2000

[8] 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

[9] Haitham Akkary, Michael A. Driscoll, *A Dynamic Multithreading Architecture*, Proceedings of the 31st MICRO, pp.226-236, Nov. 1998

[10] Pedro Marcuello, Jordi Tubella, Antonio Gonzalez, *Value Prediction of Speculative Multithreaded Architecture*, Proceedings of the 32nd MICRO, pp.230-237, Nov. 1999