ELectronics EXpress

IEICE Electronics Express, Vol.9, No.4, 290–295

Selective restart of threads for efficient thread-level speculation on multicore architecture

Sungjae Lee and Inhwan Lee[a)]
School of Electrical and Computer Engineering, Hanyang University, Hangdang-Dong 17, Sungdong-Gu, Seoul 133–791, Korea
a) *ihlee@hanyang.ac.kr*

Abstract: An efficient recovery method for thread-level speculation (TLS) is proposed. The method tracks the inter-thread data dependence as a method for identifying those threads that are obviously unaffected by a data dependence violation. The method is simple to implement. Still, the simulation results using benchmark applications show that the method can significantly reduce the number of unnecessary thread restarts and consequently improve the performance of TLS. Specifically, when compared with the baseline TLS, TLS with the proposed method is 2.3 times faster for *IS*, 1.7 times faster for *equake*, and 3.5 times faster for *mcf* with the use of 64 cores. With the method, the performance of TLS increases steadily up to 64 cores for *IS*, *equake*, and *mcf*, while the speedup of the baseline TLS starts to saturate at 8 or 16 cores.

Keywords: multicore architecture, thread-level speculation, selective restart
Classification: Electron devices, circuits, and systems

References

[1] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," *Proc. ASPLOS*, San Jose, USA, pp. 58–69, Oct. 1998.
[2] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 866–880, Sept. 1999.
[3] J. G. Steffan and T. C. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," *Proc. HPCA*, Las Vegas, USA, pp. 2–13, Jan. 1998.
[4] M. K. Prabhu and K. Olukotun, "Exposing speculative thread parallelism in SPEC2000," *Proc. PPoPP*, Chicago, USA, pp. 142–152, June 2005.
[5] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, "Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation," *Proc. ICS*, Cambridge, MA, USA, pp. 179–188, June 2005.

© IEICE 2012
DOI: 10.1587/elex.9.290
Received December 28, 2011
Accepted January 12, 2012
Published February 25, 2012

ELectronics EXpress

IEICE Electronics Express, Vol.9, No.4, 290–295

1 Introduction

Thread-level speculation (TLS) has been proposed to take advantage of multiple cores in chip multiprocessors without burdening programmers to explicitly parallelize sequential applications [1, 2, 3, 4]. With TLS, parts of a sequential program, such as loops and subroutine calls, are divided into speculative threads so that they can run on multiple cores. Data dependences among threads are not analyzed a priori, hoping that dependence violations do not occur. However, violations do occur, and TLS employs special hardware to detect them at runtime.

In conventional TLS, when a violation is detected in a speculative thread, the thread is restarted. At the same time, all threads which are more speculative than the violated thread are restarted as well because they might have been affected by the violation. However, if some of these threads did not actually use the data affected by the violation, we are restarting more threads than necessary, which means performance loss. Such an inefficient recovery can cause the efficiency of TLS to drop as the number of cores increases. Note that most studies of TLS have focused on a small number of cores.

This paper proposes an efficient recovery method which reduces the number of unnecessary thread restarts by identifying those threads that are obviously unaffected by a violation. The method is simple to implement and can significantly improve the performance of TLS especially on many-core architecture. Nested TLS has been proposed to exploit the additional parallelism that exists in nested loops or subroutine calls by supporting out-of-order thread spawning [5]. This study does not assume nested TLS.

2 Selective restart of threads

Fig. 1 illustrates the problem in conventional TLS. The threads represent speculative executions of different iterations of a single loop. When Thread #0 performs "st X," Thread #1 realizes that the speculative "ld X" it has performed violates the read-after-write data dependence. That is, Thread #1 is the violated thread, and it must be restarted. Given the violation, in conventional TLS, all threads which are more speculative than the violated thread (that is, all successor threads from Thread #2 through Thread #$n-1$) are restarted for recovery because they might have been affected. However,

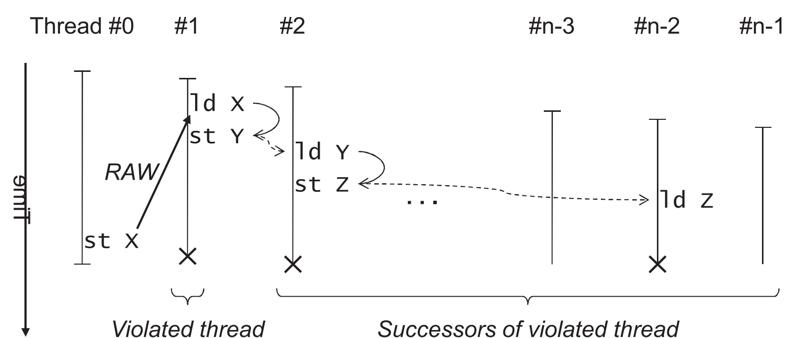

Fig. 1. Data dependence violation and recovery in TLS

© IEICE 2012
DOI: 10.1587/elex.9.290
Received December 28, 2011
Accepted January 12, 2012
Published February 25, 2012

ELectronics EXpress

if we assume that all data dependence and data forwarding between threads are shown in Fig. 1, it is sufficient to restart only Thread #2 and Thread #$n-2$ to recover from the violation. Clearly, there can be many unnecessary thread restarts in conventional TLS.

For efficient recovery, we need to identify and restart only those threads that are really affected by a violation. This strategy is called selective restart of threads (SRT) in this paper. To facilitate SRT, we have to track two types of dependences: inter-thread and within-thread dependences. Within-thread dependences are generated by the dependences between instructions within a thread. For example, in Fig. 1, data "Y" is within-thread dependent on data "X" in Thread #1. Inter-thread dependences are created by data forwarding between running threads. In Fig. 1, Thread #2 is inter-thread dependent on Thread #1.

Tracking the inter-thread dependence is relatively straight-forward and can be done by monitoring data forwarding between threads. However, tracking the within-thread dependence is quite complex. To do that, a thread must remember the address of each speculative load. It also has to maintain the information on the set of registers and memory locations, which are affected by each speculative load, by performing data flow analysis during instruction execution. Further, a thread must consider that the behavior of a thread restarted due to a violation can be different from its previous execution due to control dependency. Tracking both inter-thread and within-thread dependences completely reveals the effect of a violation on running threads. Therefore, SRT can restart only those threads that are really affected by a violation. This means that SRT can improve the performance of TLS by removing unnecessary thread restarts that exist in conventional TLS. While implementing SRT is feasible, it is expensive. Specifically, implementing SRT means a significant amount of hardware data structure (of the order of kilobytes per core) and changes in core design to implement the associated control logic.

Given the complexity of SRT, we observe that most of the data forwarded from restarted threads under SRT are actually affected by the violation. This observation suggests that we may skip tracking the within-thread dependence and track only the inter-thread dependence. Such a sub-optimal strategy is called $\mathrm{SRT_{inter}}$ or simply $\mathrm{SRT_i}$. Since $\mathrm{SRT_i}$ does not track the within-thread dependence, given a violation, it cannot accurately tell whether a certain data forwarding is affected by the violation. Therefore, $\mathrm{SRT_i}$ treats all data forwarding between threads as potentially dangerous. For example, under $\mathrm{SRT_i}$, given that Thread #1 has forwarded data to Thread #2, we unconditionally restart Thread #2 when Thread #1 restarts due to a violation. More generally, under $\mathrm{SRT_i}$, a speculative thread is restarted if it has received data from its predecessor thread being restarted due to a violation. There can be some unnecessary thread restarts under $\mathrm{SRT_i}$ because $\mathrm{SRT_i}$ is only an approximation of SRT. However, $\mathrm{SRT_i}$ is very simple to implement.

To track the inter-thread dependence for $\mathrm{SRT_i}$, all a thread has to do is to maintain a small hardware data structure called the list of forwarding

© IEICE 2012
DOI: 10.1587/elex.9.290
Received December 28, 2011
Accepted January 12, 2012
Published February 25, 2012

ELectronics EXpress

IEICE Electronics Express, Vol.9, No.4, 290–295

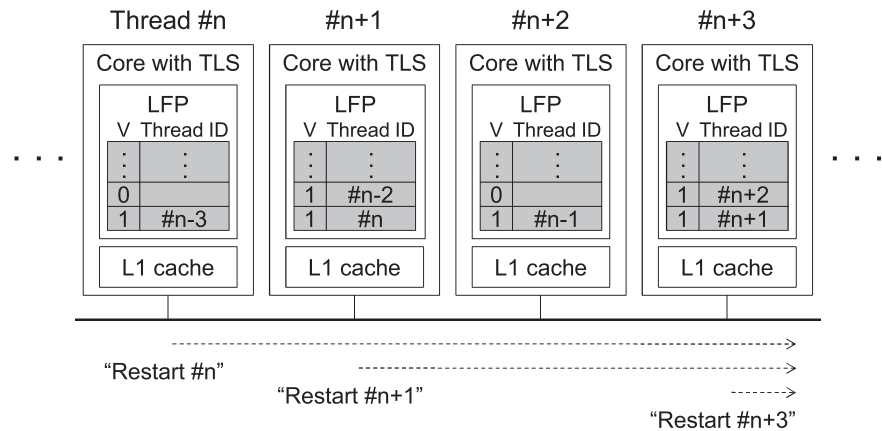

Fig. 2. Recovery from violation under SRT_i

predecessors (LFP). Fig. 2 illustrates a snapshot of LFPs when Thread $\#n$ detects a violation. The number of entries in LFP is equal to the number of cores. An LFP entry consists of valid bit and thread identification. A thread clears its LFP when it is restarted or committed. When a thread loads data from one of its predecessor threads, an entry of LFP is allocated to record the identification of the predecessor thread unless the thread already exists in LFP.

Let's examine the recovery process under SRT_i using Fig. 2. Given the violation in Thread $\#n$, the thread sends a restart message to its successors and restarts. Given the message, Thread $\#n+1$ looks up its LFP and realizes that it has received data from Thread $\#n$, while Threads $\#n+2$ and $\#n+3$ ignore the message based on their LFPs. So Thread $\#n+1$ sends out a message and restarts. On receiving the message from Thread $\#n+1$, Thread $\#n+3$ sends out a message and restarts because it has received data from Thread $\#n+1$. Such a chain reaction is repeated until no more thread restart is necessary.

3 Experimental results

We build an execution-driven cycle-accurate simulator which models a shared-bus chip multiprocessor with multiple cores and fully hierarchical memory system. We implement the baseline TLS architecture in [4] into our simulator. We also implement SRT and SRT_i. Note that the only difference between this work and the baseline TLS is the recovery scheme. We use a simple core model with a single pipeline. Both the L1 instruction and data caches in each core are 4-way 4 KB caches with 1 cycle of latency. We employ the write-invalidate cache coherence protocol. The 8-way 1 MB L2 cache is shared by all cores and has 5 cycles of latency. The main memory has 100 cycles of latency. We assume the latency of 8 cycles for spawning a thread, 12 cycles for committing a thread, and 7 cycles for restarting a thread.

We select four benchmark applications with varying degrees of data dependence. *IS* is a benchmark from NPB3.3-SER; *equake* is a floating point benchmark from SPEC2K and *mcf* is an integer benchmark from SPEC2K;

© IEICE 2012
DOI: 10.1587/elex.9.290
Received December 28, 2011
Accepted January 12, 2012
Published February 25, 2012

ELectronics EXpress
IEICE Electronics Express, Vol.9, No.4, 290–295

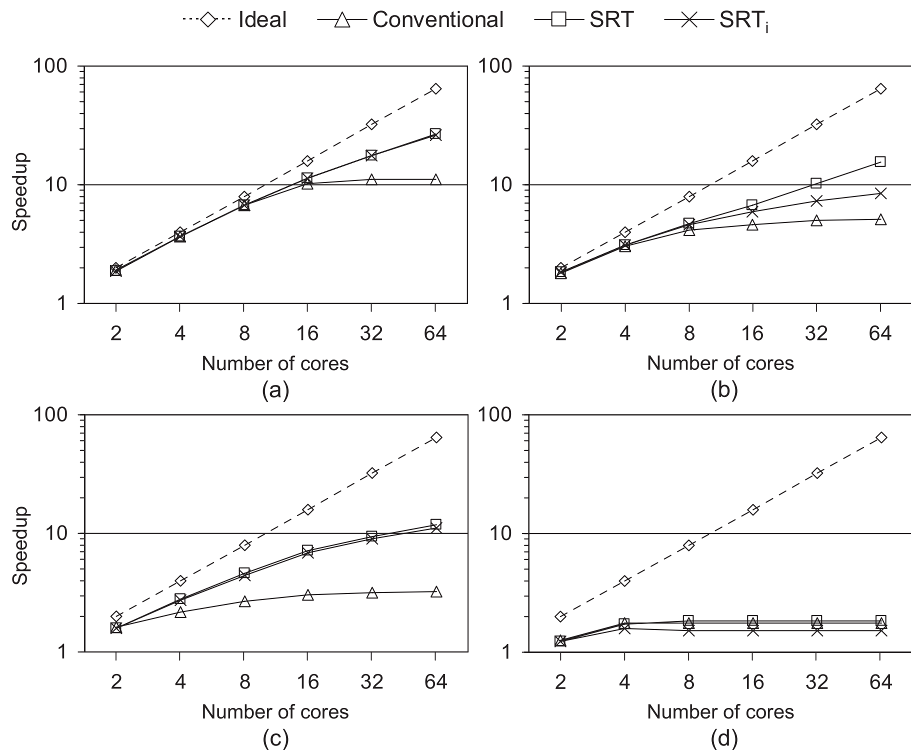

Fig. 3. TLS speedup for (a) *IS*, (b) *equake*, (c) *mcf*, and (d) *compress*

compress is an integer benchmark from SPEC95. Data dependence between threads is weak in *IS*, moderate in *equake* and *mcf*, and strong in *compress*. We then manually select the loops in the benchmarks and parallelize different loop iterations as speculative threads using the parallelization techniques discussed in [4]. Using the simulator and the parallelized benchmarks, we finally evaluate the performance of TLS under three recovery strategies: conventional recovery, SRT and SRT_i.

Fig. 3 shows the speedup achieved by TLS with the three recovery strategies. The dotted line at the top represents the ideal speedup. SRT represents the performance of TLS under perfect recovery. The difference between SRT and 'conventional' is due to unnecessary thread restarts in conventional TLS; the gap between 'ideal' and SRT reflects the wasted time due to data dependence violation and recovery; the difference between SRT and SRT_i represents the performance loss caused by not tracking the within-thread dependence.

SRT_i is very simple to implement. Still, it can significantly reduce the number of unnecessary thread restarts and consequently improve the performance of TLS. Fig. 3 shows that, when compared with the baseline TLS, TLS with SRT_i is 2.3 times faster for *IS*, 1.7 times faster for *equake*, and 3.5 times faster for *mcf* with the use of 64 cores. As for *IS* and *mcf*, SRT_i is as effective as SRT. As for *equake*, SRT_i is effective but not as good as SRT. Close examination of data reveals that most of the threads restarted due to a violation under SRT_i have actually received and used the data affected by the violation in *IS* and *mcf*, which explains why SRT_i is as effective as SRT.

© IEICE 2012
DOI: 10.1587/elex.9.290
Received December 28, 2011
Accepted January 12, 2012
Published February 25, 2012

IEICE Electronics Express, Vol.9, No.4, 290–295

In contrast, the threads restarted due to a violation under SRT_i have often not used the affected data in *equake*. So SRT can perform better than SRT_i. As expected, regardless of the recovery strategy used, TLS is not effective for *compress*. Due to strong data dependence between threads, unnecessary thread restarts are uncommon in *compress*.

Finally, note that the performance of TLS with SRT_i increases steadily up to 64 cores for *IS*, *equake*, and *mcf*, while the speedup of the baseline TLS starts to saturate at 8 or 16 cores.

4 Conclusion

This paper presents a method, called SRT_i, to efficiently recover from a data dependence violation in TLS. By tracking the inter-thread data dependence, SRT_i restarts a speculative thread only when the thread has received data from its predecessor thread being restarted due to a violation. The method is very simple to implement. The simulation results using benchmark applications show that, unless the data dependence between threads is very strong, the method can significantly reduce the number of unnecessary thread restarts and consequently improve the performance of TLS especially on many-core architecture. Evaluating the effectiveness of selective recovery under out-of-order thread spawning will be an interesting future work.

© IEICE 2012
DOI: 10.1587/elex.9.290
Received December 28, 2011
Accepted January 12, 2012
Published February 25, 2012