2011 UKSim 5th European Symposium on Computer Modeling and Simulation

Applying Formal Verification to A Cache Coherence Protocol in TLS

Xin Lai, Cong Liu, Zhiying Wang
School of Computer, National University of Defense Technology
NUDT
Changsha, Hunan Province, China
laixin1982@gmail.com, congliu@nudt.edu.cn, zywang@nudt.edu.cn

***Abstract*—Current hardware implementations of TLS (thread-level speculation) in both Hydra and Renau's SESC simulator use a global component to check data dependence violations, e.g. L2 Cache or hardware list. Frequent memory accesses cause global component bottlenecks. In this paper, we propose a cache coherence protocol using a distributed data dependence violation checking mechanism for TLS. The proposed protocol extends the traditional MESI cache coherence protocol by including several methods to exceed the present limits of centralized violation checking methods. The protocol adds an invalidation vector to each private L1 cache to record threads that violate RAW data dependence. It also adds a versioning priority register that compares data versions. Added to each private L1 cache block is a snooping bit which indicates whether the thread possesses a bus snooping right for the block. The proposed protocol is much more complicated than the traditional MESI protocol and hard to be completely verified only through simulation. So we applied formal verification to the proposed cache protocol to confirm its correctness. The verification result shows that the proposed protocol will function correctly in TLS system.**

Keywords-TLS; Cache Coherence Protocol; Snooping Ring; Invalidation Vector; Formal Verification

I. BACKGROUND

As Moore's Law [1] predicts, hardware is becoming progressively smaller and execution times quicker. The current hardware world is dominated by the multi-cores or many-cores. While the effort to design and implement CMP (Chip Multi-Processor) has been alleviated, there is a heavy mission undertaken by the compiler or programmer.

Thread level speculation (TLS) has been proposed to help the programmers to make full use of the abundant computing resources. TLS divides a sequential code into pieces (threads) and executes them in parallel. TLS optimistically assumes that the sequential semantics will not be violated. The architecture transparently detects any data dependence violations as the threads execute. Then if an offending thread read too early, it will be squashed and returned to a previous non-speculative correct state. Thus, TLS can improve performance by exploiting the Thread-Level Parallelism (TLP).

A TLS technique can be installed in ether hardware or software. In software, every memory access is wrapped by a very short piece of codes which collects access information. When memory access is a write, the wrapper would detect a violation [5].In hardware, many architecture features are available to enable TLS [2][4].

To provide the desired memory behavior, the data speculation hardware must [2]:

1. Forward data between parallel threads.
2. Detect when reads occurs too early (RAW hazards)
3. Safely discard speculative state after violations.
4. Retire speculative writes in the correct order (WAW hazards).
5. Provide memory renaming (WAR hazards).

Our paper is organized as follows. First, we introduce the design considerations and the principle of the proposed protocol in Section 2. Then we discuss implementation details including, but not limited to, hardware alteration, cache block state transition, and cache miss events handling. Protocol formal verification is reported in Section 3. We brief related works in Section 4. Finally we draw conclusions in Section 5.

II. RELATED WORKS

Thread Level Speculation Systems. Many systems have implemented the TLS mechanism. Some are supported directly with architectural features [2] [3] [4], while others use software methods [5][6]. The software ones complement our work.

Speculative Versioning Caches [4]. SVC combines the speculative versioning with cache coherence based on the snooping bus. SVC adds some useful bits to each cache line, such as bits of commit, load, store, stale etc., that is somewhat similar to our work. Our work eliminates ARB's latency and bandwidth problems and provides a more efficient performance for distributed caches and in all of the above methods. Our work fits the SMP systems well, and not only for the current CMP. The most precious values of this research is the model analyzing the speculative action between issued load and store instructions.

The Stanford Hydra CMP [2][7][8]. Hydra provides speculative write buffers for each speculative thread in the on-chip L2 cache, but only one thread owning the writing token is permitted to retire. These embedded buffers have many advantages. First, they can forward data between parallel threads. Second, because of the adjacent buffers, they can detect when reads occurring too early. Third, they discard speculative state after violations or commit safe data to the constant state in the L2 cache directly. But if data dependence among threads occurs frequently, this method will greatly will greatly increase the latency for loading other threads' data, and the detection time increases dramatically.

Renau's SESC Simulator. The SESC simulator [9] supports TLS via some microarchitecture features such as splitting timestamp intervals, immediate successor list, and dynamic task merging [3]. SESC detects memory-based data dependence violations using a method similar to the hardware list method. Each thread's memory access (load and store) maintains a private list in the L1 data cache. All these accesses are gathered into a common list in the L2 cache and sequenced by the access's address. This list is used to detect violations. When memory is not accessed much, TLS system performs well. However, when the access is frequent, the common list bottlenecks all the parallel threads.

STAMPede (Single-chip tightly-coupled architecture for multiprocessing) [STAMPede] is a speculative multi-thread executing architecture [10]11][12][13][14]. It extends the traditional MESI protocol by adding a speculative loading [SL] bit and a speculative modify [SM] bit to each cache block. During execution, it monitors events from the bus and the local core, maintains the states for every cache block, records memory access information for threads, and checks for violations.

III. TLS SUPPORTING CACHE COHERENCE

In this section, the control mechanism for the proposed protocol is demonstrated. Our protocol is based on the traditional snooping bus-based MESI coherence protocol. Four types of events require cache coherence controller support. Events and corresponding actions are organized as Table 1 shows. We use the teams "exposed read" or "exposed write" rather than speculative read or write.

TABLE I. TLS CACHE EVENTS AND ACTIONS

Events	Actions
Exposed read	Read data from main memory or shared cache into private cache
Exposed write	Write data to buffer before commit
Commit	Write back thread-created buffered data to main memory or shared cache ,and invalidate other thread that depends on its executing results
Squash	Invalidate thread-created buffered data

A. Brief introduction to the proposed protocol

We use the following management policies for speculative thread to simplify cache coherence protocol.

1. In-order speculative threads spawning and committing;
2. Speculative threads reside in the core before committing. Other threads are unable to swap an executing thread out of its host core, even if the threads have lower speculation degree.

As a result of these management polices exposed reads and exposed writes may only run against data dependences present in the original sequential program. Cache coherence controllers must record exposed reads immediately following each exposed write in order to maintain RAW data dependence. Speculative threads reside in the core before committing. Other speculative writing thread cannot be scheduled to the same core. Each speculative thread has a private L1 cache which buffers modified data and removes WAW data dependence. There may be different data versions in L1 Caches but only the most recently modified data version has the right to monitor bus events and be forwarded to other threads. We propose a solution for this problem in the following section. All data in a thread share the same version priority. TLS runtime establishes an initial version priority for each speculative thread when the thread is scheduled. TLS updates the threads version priority as other threads successfully commit.

The main ideas of the proposed protocol are as follows.

1. An invalidation vector is added to each private L1 cache in order to identify cores that violate RAW data dependence.
2. A version priority register is added to each core to resolve data versioning conflicts and gives a less speculative thread with a higher version priority. When a read miss happens, the coherence controller places the version priority value on the bus.
3. A speculative executing mode is added to the L1 cache to prevent the version priority reversed phenomenon from occurring.
4. A snooping bit is added to each L1 cache block in order to maintain WAW and RAW data dependence. The protocol must insure that only the "freshest" data version will have the snooping bit set. The "freshest" data version is the most recently modified data copy.
5. An exposed read (ER) and an exposed write (EW) bits are added to each L1 cache block.

B. Detail implementation of the proposed protocol

In this section, we introduce a modification to the traditional cache coherence protocol and discuss how the individual operation listed in Table I is performed. Our protocol targets providing data dependence maintenance, data forwarding, memory renaming and efficient dependence violation detection, without altering the current cache implementation method dramatically. In the current CMP implementation, each core has a L1 cache and a shared L2 cache or L3 cache. To discriminate between non-speculative and speculative read & write, we add EW (exposed write) and ER (exposed read) bits to each L1 cache block. In order to differentiate between non-speculative execution and speculative execution, we introduced a speculative execution mode to the private L1 cache. We divided the pre-commit thread life cycle into 3 stages: execution, pre-commit and optional re-execute. Also we added 3 modes to the private L1 cache: speculative mode, re-execute mode and pre-commit mode. The re-execute mode and pre-commit modes are sub-mode of the speculative execution mode that can only be activated when the L1 cache is in the speculative mode. When a thread is invalidated by other threads, TLS runtime resets the L1 Cache to re-execute. When a thread is executed and ready to commit, TLS runtime changes the status of the L1 cache to pre-commit mode.

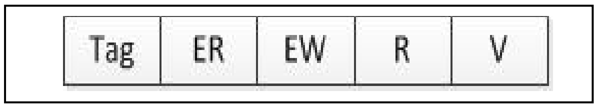

Figure 1. L1 Cache block organization. ER: exposed read EW: exposed write R: Snooping ring flag V: valid

As we noted in the previous section, we have added three bits to each L1 cache block. Figure 1 shows the organization of private L1 cache block. We have also introduced an invalidation vector and a version priority register to each core.

1. Each cache block maintains an ER bit as shown in Figure 3. The ER bit is set when a thread reads data from main memory or the shared cache, while in the speculative execution mode. If a thread reads data before less speculative threads store it, RAW dependence occurs and the thread must be invalidated and re-executed.

2. Each cache block maintains an EW bit as shown in Figure 3. The EW bit is set when a thread modifies data which has been loaded into the L1 cache, while in the speculative execution mode. Different threads may modify the same data before they write data back to the shared cache or the main memory. This event leads to different data versions residing in different L1 caches. We discriminate between the "freshest" data version and non-freshest data version. Only the "freshest" modified data can be forwarded to other threads.

3. Each cache block maintains an R [snooping ring] bit. If a thread writes data to an L1 cache for the first time and that L1 cache is not in the re-execute mode, than the write operation triggers an exposed write miss event. The thread is than awarded the snooping rights. This R bit cooperates with the EW bit to provide for RAW data dependence violation detection and data forwarding.

4. Each core maintains an invalidation vector to provide quick thread invalidation. Each bit is mapped to one core.

In addition to traditional requests from the local processor or bus, several new requests are introduced that the coherence controller must address. The new requests include:

1. An exposed write miss from the local processor while cache block is invalid. The cache coherence controller addresses this event by placing an exposed write miss notation on the bus and changing the cache block state to EWR.

2. An exposed read miss from the local processor while cache block is in the invalid status. The cache coherence controller addresses this event by placing an exposed read miss on the bus and changing the cache block state to ER.

3. An exposed write miss from the local processor while the cache block is in the ER status. The cache coherence controller addresses this event by placing an exposed write miss on the bus and changing the cache block state to EWR.

4. A write commit from the local processor while the cache block is in the EWR or EW state. The cache coherence controller addresses this event by invalidating other threads according to the priority established in the invalidate vector and changes the cache block state to invalid.

5. An exposed write miss from the bus while the cache block is in the EWR status. The cache coherence controller addresses this event by clearing the R bit in the cache block.

6. An exposed read miss from the bus while the cache block is in the EWR status. The cache coherence controller addresses this event by setting the corresponding bit in the invalidate vector according to the core ID.

The state machine for the proposed protocol is shown in Figure 2, 3.

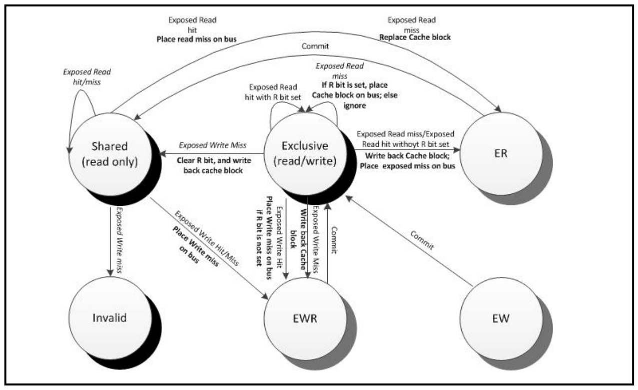

Figure 2. State Transition between Speculative States

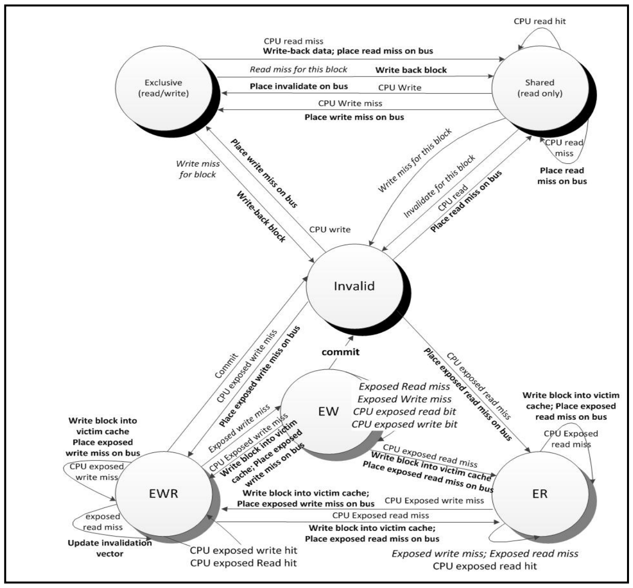

Figure 3. State Transition between Speculative States and non-speculative States

Exposed Write When a speculative thread executes for the first time, a write to any address causes an exposed write miss event. The coherence controller detects the write miss and places the exposed write miss event on the bus to inform any other processor to give up its bus snooping rights. Simultaneously, the coherence controller sets the EW and R bits to change the corresponding cache block to the EWR state. While in the EWR state, any exposed read or write hits from the local processor will be ignored.

If a cache block is in the EWR state, the coherence controller updates the invalidation vector by setting the bit according to the core ID from the bus if the address of a read miss from the bus matches the cache block address field. A write miss from the bus forces the cache coherence controller to clear the R bit in the cache block. When a cache block loses its snooping ring, the coherence controller changes its state to EW. In this state, the cache coherence controller ignores all requests from either the local processor or bus.

When a speculative thread becomes non-speculative, the cache coherence controller writes cache blocks with the EW bit set back to the shared cache or main memory and changes their states to exclusive. If the invalidation vector is not empty, the cache coherence controller must send an invalidation signal to any core whose invalidation bit has been set in the invalidation vector. Otherwise the controller simply clears the cache block and changes its status to invalid. After committing, the invalidation vector must be cleared to execute the next speculative thread.

There is a significant difference between the proposed protocol and the traditional MESI protocol in managing exposed read and writes misses caused by address conflicts. The traditional MESI protocol only supports non-speculative execution, any modification to data can be written back to the shared cache or main memory and cache lines can be replaced with new data. TLS is unable to write data back unless threads become non-speculative. When an address conflict occurs, a cache block with the EW bit set cannot be written back nor can it be discarded. Current cache implementations provide functions to resolve address conflicts, such as locking a cache line or mapping the new data to other cache blocks. We use a victim cache to hold the conflicting blocks. Present TLS implementation threads are always either “while” loop or “for” loop and does not contain sufficient data to trigger address conflict. Other speculative threads are not permitted to be scheduled to the same core before the active thread commits. The chance of inter-threads address conflict is negligible.

Exposed Read Handling an exposed read is much simpler than handling an exposed write. The coherence controller places a read miss together with the version priority and core ID on the bus. When data from a remote core is ready, access to the shared cache or main memory is aborted. After a cache line has been filled with data, the coherence controller sets the ER bit to change the block status to ER.

When a cache block is in the ER status, the coherence controller ignores all requests from the local core except read misses or write misses. The coherence controller handles a read miss from the local processor by replacing the cache block with new data and keeps the cache block in the ER status. Upon receiving a write miss request, the coherence controller not only replaces the cache block with new data but also clears the ER bit and sets the EW and R bits.

Invalidation Optimization The proposed protocol works well if no thread is invalidated during execution. Adhering to the 1st thread management policy, if a thread gets the snooping ring, it will record threads that violate the RAW data dependence when relinquishing the snooping ring. The coherence controller doesn’t need to monitor read misses on the bus and record threads a second time if a thread lacking the snooping ring is invalidated and re-executes. However the proposed protocol does not distinguish between a write miss from a thread running for the first time nor does it distinguish between a write miss from a re-executing thread. A write miss from a re-executing thread that lacks the snooping ring will cause a newer, or more speculative, thread to relinquish the snooping ring and the cache coherence controller will record any read misses on the bus belonging to the more speculative thread. This event is named ‘priority reverse’. When a thread attempts to commit, TLS runtime will invalidate wrong threads. A re-execute mode was added to the L1 cache in order to distinguish between the two different types of write miss.

When a thread is invalidated and re-executes, TLS runtime changes the L1 cache to the re-execute mode. When the L1 cache is In the re-execute mode, any write miss from the local core that happens to the data cache block with the EW bit set will not be placed on the bus and obtain the snooping ring.

When a thread has executed and is ready to commit, the cache coherence controller continues to monitor bus activity. If a cache block is with the R bit set, the cache coherence controller will detect RAW data dependence violation and update the invalidation vector. The thread may forward data to other threads without canceling those threads executing when it commits. System performance is degraded by re-launching a thread because it a wastes a bit.

After a thread has run, TLS runtime establishes the pre-commit bit for the L1 Cache. If a low version priority read miss request from the bus hits a cache block with the R bit set, the coherence controller will place data on the bus if the L1 cache is in the pre-commit mode. The controller then updates the invalidation vector.

IV. Protocol Formal Verification

In order to apply formal verification to the proposed cache coherence protocol, we introduced the following symbols.

TABLE II. NOTATION

Variable	**Description**
$S(addr_i)$	The state of the cache line in the core i with the address field equaling to $addr$
$LDEI(addr_i)$	The initial exposed load operation launched by the core i for data with the physical address $addr$
$STEI(addr_i)$	The initial exposed store operation launched by the core i for data with the physical address $addr$
$Time(event)$	The event happening time
$Spec(addr_i)$	The speculation degree of the thread

Variable	Description
	executed in the core i that accesses the address $addr$
$Ring(addr_i)$	The core i pose the bus snooping ring for the address $addr$
$InvVec(i)$	Invalidation vector in the core i

$$S(addr_i) \in \{Invalid, Exclusive, Shared, EWR, EW, ER\}$$

The state transition can be expressed by the following equations. From the above discussion, we know that the proposed TLS mechanism will keep memory accesses operation in order, which means if any memory access is behind other memory accessed it will not be scheduled to be executed in advance.

Theorem 1. For any two exposed load operations for the same address $addr$ in different active speculative threads. If there are other exposed load operations between them in original application, the later executed load operation in the original sequential program will get the bus snooping ring right for the address $addr$ from the former one.

$$\forall addr, i \neq j, \quad Spec(addr_i) < Spec(addr_j), Ring(addr_i)$$

frm: $_{STEI(addr_j)}$

$$\Rightarrow \quad Ring(addr_j)$$

prof:

$$\forall k, k \neq i \neq j \cap !\exists h, Spec(addr_i) < Spec(addr_h) < Spec(addr_k)$$
$$\because !\exists h, Spec(addr_i) < Spec(addr_h) < Spec(addr_k)$$
$$\therefore Spec(addr_h) > Spec(addr_k) \vee Spec(addr_h) < Spec(addr_i)$$
$$Spec(addr_h) > Spec(addr_k)$$
$$\Rightarrow Time(STEI(addr_h)) > Time(STEI(addr_k))$$
$$Spec(addr_h) < Spec(addr_i)$$
$$\Rightarrow Time(STEI(addr_h)) < Time(STEI(addr_i))$$
$$\therefore !\exists h, Time(STEI(addr_i)) < Time(STEI(addr_h)) < Time(STEI(addr_k))$$
$$\therefore Ring(addr_i) \xrightarrow{STEI(addr_k)} Ring(addr_k)$$
$$\text{Re}cursely, Ring(addr_k) \xrightarrow{STEI(addr_l)} Ring(addr_l)$$
$$\cdots Ring(addr_p) \xrightarrow{STEI(addr_j)} Ring(addr_j)$$

Because L1 cache will keep snooping ring even the speculative is squashed. The theorem 1 insures that the proposed protocol will keep the WAW sematic in original program. The snooping will not be transferred from the more speculative thread back to the less speculative thread.

Theorem 2. For any exposed load operations that follow an exposed read operation for the same address $addr$ in different active speculative threads. If there are other exposed load operations between them in original application, the core i will catch all exposed read operations before the speculative thread gives up the bus snooping ring for the address $addr$.

frm:

$$\forall addr, i, j, k, STEI(addr_i), LDEI(addr_j), STEI(addr_k), i \neq j \neq k$$

$$\begin{cases} Spec(addr_i) < Spec(addr_j), Ring(addr_i) \\ Spec(addr_k) < Spec(addr_i) \vee Spec(addr_k) < Spec(addr_k) \end{cases}$$

$$\Rightarrow j \in InvVec(i)$$

prof:

$$\begin{aligned} & Spec(addr_k) < Spec(addr_i) \\ & \Rightarrow Ring(addr_k) \xrightarrow{STEI(addr_i)} Ring(addr_i) \end{aligned} \quad (1)$$

$$\begin{aligned} & Spec(addr_k) < Spec(addr_k) \\ & \Rightarrow Time(STEI(addr_k)) > Time(LDEI(addr_j)) \end{aligned} \quad (2)$$

$$\therefore !\exists k \; Ring(addr_i) \xrightarrow{STEI(addr_k)} Ring(addr_k) \quad (3)$$

Combining E.q (1) with (3), we can get the following equation.

$$Ring(addr_i) \xrightarrow{LDEI(addr_j)} j \in InvVec(i).$$

Because L1 cache will keep snooping ring even the speculative is squashed. The theorem 2 insures that the proposed protocol will keep the RAW sematic in original program. From the theorem 2 verification process, we notice the following result. If a speculative store does not fall between another speculative store and a speculative load, it will not try to get the snooping ring before the speculative load happens. So the WAR sematic will be kept, a speculative will not be squashed by an "inappropriate" thread.

Learn from the above verification process, we confirm that the proposed cache coherence protocol will violate data dependences existing in original application.

V. Conclusion

Memory system design in TLS systems is very important. Tradeoffs between hardware complexity and functionality are essential. In this paper we proposed a distributed data dependence violation detection cache coherence protocol for TLS. Data dependence violation detection is done by each core, not by a global component as the SESC simulator does, nor by altering cache functionality as transaction memory [TM] does. Private L1 cache provides a perfect memory location for data renaming. The greatest advantage of the proposed protocol is that it does not need to broadcast write events while executing store instructions. It uses a snooping ring together with an invalidation vector to address RAW dependence violation detection issues. The correctness of the proposed protocol is confirmed through formal verification.

Data forwarding is important feature in TLS system, but our protocol does not include the feature because of thread squash. It is also fruitless effort to forward data between speculative threads when intermediate data is not the final result. It is our next target to provide an effect data forwarding method in TLS system when keeping the TLS system implementation as simple as possible.

Reference

[1] G. E. Moore, "Cramming More Components onto Integrated Circuits," Electronics, vol. 38, April 1965.

[2] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen,and K. Olukotun. The Stanford Hydra CMP. IEEE Micro Magazine, March-April 2000.

[3] 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. In International Conference on Supercomputing (ICS), pages 179–188, June 2005.

[4] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In HPCA 4, February 1998.

[5] P. Rundberg and P. Stenström. An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors. The Journal of Instr.-Level Par., 1999.

[6] C. J. F. Pickett and C. Verbrugge. Software Thread Level Speculation for the Java Language and Virtual Machine Environment. In Lang. Comp. Par. Comp. (LCPC), Oct 2005.

[7] L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), Oct. 1998

[8] K. Olukotun, L. Hammond, and M. Willey, "Improving the Performance of Speculatively Parallel Applications on the Hydra CMP," Proc. 1999 Int'l Conf. Supercomputing(ICS), June 1999.

[9] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[10] Steffan J.G, Colohan C.B, Mowry T.C., Architectural Support for Thread-Level Data Speculation[R], Technical Report CMU-CS-97-188, School of Computer Science, Carnegie, Mellon University, 1997.

[11] Steffan J.G., Mowry T.C., The Potential for Thread-Level Data Speculation in Tightly-Coupled Multiprocessor [R]., Technical Report CSRI-TR-350, Computer Science Research Institute, University of Toronto, 1997.

[12] Steffan J.G., Colohan C.B., Zhai A., et al., A Scalable Approach to Thread-Level Speculation [C], Proceedings of the 27th Annual International Symposium on Computer Architecture, 2000.

[13] Steffan J.G., Colohan C.B., Zhai A., et al., Improving Value Communication for Thread-Level Speculation [C]. High-Performance Computer Architecture , 2000. Proceedings. Eighth International Symposium, 2000,P. 65-75.

[14] Steffan J.G., Colohan C.B., Zhai A. et al., The STAMPede Approach to Thread-Level Speculation [J], ACM Transaction on Computer Systems, 23, issue3, 2005, P.253-300.