# Complexity Analysis of A Cache Controller for Speculative Multithreading Chip Multiprocessors

Yoshimitsu Yanagawa,† Luong Dinh Hung,†† Chitaka Iwama ,††
Niko Demus Barli,†† Shuichi Sakai†† and Hidehiko Tanaka††

Although many performance studies of memory speculation mechanisms in speculative multithreading chip multiprocessors have been reported, it is still questionable whether the mechanisms are complexity effective and worth to implement. In this paper, we perform a complexity analysis of a cache controller designed by extending an MSI controller to support thread-level memory speculation. We model and estimate the delay of logic on critical paths and additional area overhead to hold additional control bits in cache directory. Our analysis shows that for many protocol operations, area overhead occupies more than half of the total delay. This area overhead is however smaller than the delay for accessing and comparing cache tags. When the protocol operations are performed in parallel with tag comparison, the critical path latency is increased by 11%. Overall, our results show the cache controller can be implemented with cycle time of less than 22 [FO4], and can be made faster if we further pipeline the protocol operations.

## 1. Introduction

Chip Multiprocessor (CMP) architecture is becoming increasingly attractive because of its capability to exploit Thread Level Parallelism (TLP) in multithread or multiprogram workloads. However, to be fully accepted as a general purpose platform, it must also deliver high performance when executing sequential applications. In order to extract TLP from sequential applications, a technique called speculative multithreading has been proposed [4–6,8,9,11]. In speculative multithreading execution, a sequential program is partitioned into threads and speculatively executed in parallel. The threads may exhibit data or control dependencies depending on the thread execution model. Additional hardware and software supports are needed to recover the execution when the speculation failed.

Speculative multithreading architectures usually use a combination of control and data speculation to exploit TLP. One of useful speculation techniques is thread-level memory speculation, in which a thread executes memory *load* speculatively regardless the possibility that a predecessor thread may *store* into the same memory location. A number of hardware based mechanisms to support memory speculation have been proposed. These mechanisms usually extend the functionality of cache to buffer speculative state of memory and detect memory dependency violations [3–5, 9].

Although many performance studies of the memory speculation mechanisms have been reported, it is still questionable whether the mechanisms are complexity effective and worth to implement. This paper is an effort to answer this question. In this paper, we first modify MSI (Modified-Shared-Invalid) cache coherency protocol to support thread-level memory speculation. The hardware organization of the cache controller necessary to implement the protocol is then described and complexity analysis is performed. We model and estimate the delay of logic on critical paths and additional area overhead to hold additional control bits in cache directory.

We found that for a 32-kB cache with 64-byte lines, the increase in area overhead of cache directory over the original MSI cache directory is significant. Access latency to the cache directory is approximately 11.4 [FO4], whereas the logic delay of critical paths is less than 10 [FO4]. For many protocol operations, area overhead occupies more than half of the total delay. This area overhead is however smaller than the delay for accessing and comparing cache tags. Since cache directory access and protocol logic operation can be performed in parallel with cache tag access, significant increase in critical path delay can be avoided. In such a case, our analysis shows that the critical path latency is increased by 11%. Overall, our results show the cache controller can be implemented with cycle time of less than 22 [FO4]), and can be made faster if we further pipeline the protocol operations.

The rest of this paper is organized as follows. Section 2 describes baseline architecture and memory speculation model assumed in the paper, and presents related work. Section 3 explains cache controller support for memory speculation. Complexity of the cache controller is evaluated in Section 4. Section 5 concludes the paper.

## 2. Speculative Multithreading

### 2.1 Execution Model

Figure 1 depicts an organization of CMP architecture we use in this paper. It consists of four Processing Units (PU) with private register file, L1 I-Cache, and L1 D-Cache. A unified L2 cache is shared by all the PUs. Threads are scheduled to the PUs and their execution is validated by a Thread Control Unit.

In our speculative multithreading model, a sequential program is partitioned into threads at compile time. A thread is defined as a connected subgraph of

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

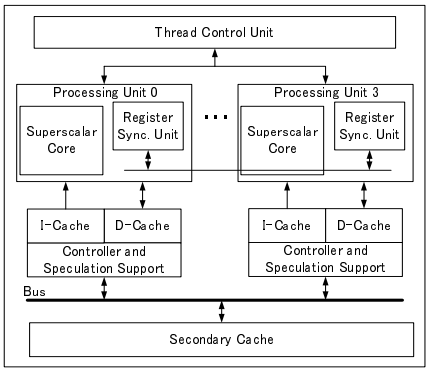


**Fig. 1** Baseline architecture

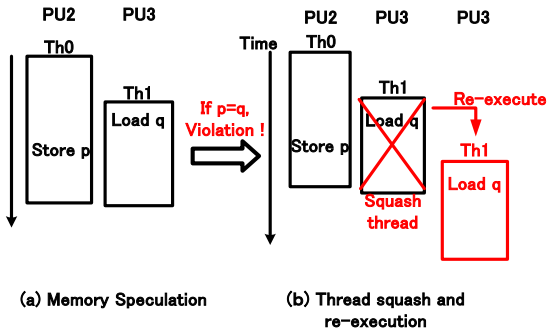


**Fig. 2** Memory Speculation

a static control flow graph with a single entry point. Thread boundaries are put at function invocations, innermost loop iterations, and at remaining points necessary to satisfy the thread definition [1]. Average thread size is 10-20 dynamic instructions.

At execution time, threads are scheduled to the PUs in a round-robin fashion. The thread model allows control and data dependences to exist between threads. The Thread Control Unit dynamically predicts threads and validates their execution. Data dependencies through register are synchronized using a combination of compiler and hardware supports. Finally, memory *loads* are executed speculatively. In case a violation occurs, the violating thread and all its successors are flushed and restarted. The mechanism of memory speculation support is the main subject of this paper, so we present a more detail explanation in the next subsection.

### 2.2 Memory Speculation

Figure 2(a) shows an example of memory speculation. Two threads Th0 and Th1 are executed concurrently in the PU 2 and PU 3. The speculation level of Th1 is higher than Th0. Without memory speculation, the load instruction in Th1 must be delayed until the execution of the store in Th0 to prevent possible memory RAW violation. In order to avoid such synchronization overhead, memory speculation allows Th1 to execute the load before the store in Th0 is executed. If the target addresses of these instructions turn out to be different, the speculation succeeds (figure 2(a)). Otherwise, Th1 as well as all its successor threads must be flushed and reexecuted (figure 2(b)).

In order to support the thread-level memory speculation, the underlying architecture must provide a mechanism to detect memory dependency violations and recover the execution. It must also provide a buffering mechanism for speculatively stored data. In this paper, we modify an MSI consistency protocol and design a cache controller for the purpose. A more detail description will be given in section 3.

### 2.3 Related Work

Many models and hardware/software supports for thread level memory speculation have been proposed [2–5, 9, 11]. Among them, [4, 5, 9] provide details on how to implement the mechanism on CMP caches.

Hydra [4] proposed an extension of cache directory to handle speculative state of the cache line. The speculation state is managed on per-line basis. A write-through policy is employed and a speculation buffer is attached to each cache to buffer speculatively stored value.

STAMPede [9] extends the traditional MESI protocol with additional states to support memory speculation. Similar to Hydra, the management of speculative state is performed on per-line basis. However, it differs in that it does not require special buffer to hold speculative memory values.

In [5], memory speculation is performed using a centralized table called Memory Disambiguation Table (MDT). MDT is located between the private L1 caches and the shared L2 cache. It records loads and stores executed on L1 caches. Memory management manages memory state on per-word basis. Since the entry of MDT is limited, memory operation of speculative threads need to be stalled if the table is full.

This work takes a middle course from the latter two approaches. Rather than using a centralized table, we extend the MSI protocol to support memory speculation similar to STAMPede. However, we choose to manage the memory state on per word basis as in [5]. It is because previous work has shown that maintaining the state on a per-line basis results in poor performance [5].

## 3. Cache Controller Support for Memory Speculation

### 3.1 Organization

We integrated a hardware mechanism into each processing unit's cache controller to support thread-level memory speculation. The organization of the controller is shown in figure 3. The controller mainly comprises four units: *state controller* that manages cache states, *cache directory* that holds cache states and speculation history, *violation detector* that detects memory violation, and *data forwarder* that controls data forwarding between PUs. The controller manages the state of memory by snooping memory events broadcasted on a shared memory bus. This

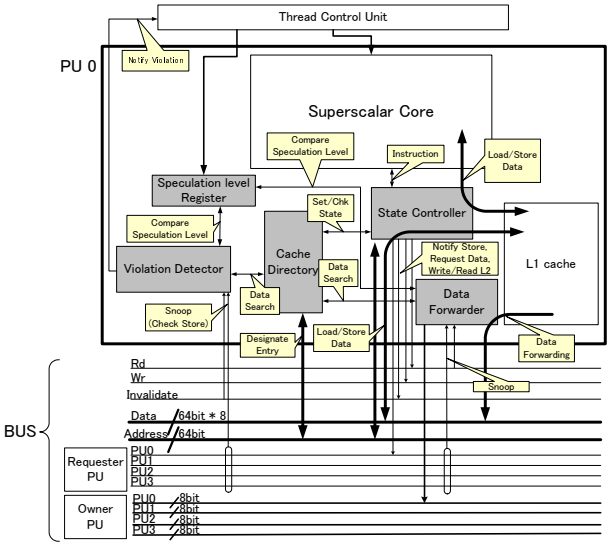


**Fig. 3** Cache controller organization.

bus includes a pair of data bus and address bus, and a number of control lines necessary to maintain the consistency of memory.

### 3.2 Coherence Protocol

Our coherence protocol allows RAW violations to occur. On the other hand, it provides support to prevent WAW and WAR violations. Data speculatively stored by a thread is temporarily held in L1 caches. The data is allowed to be committed to L2 cache only after the thread becomes a non-speculative thread, thus preventing WAW violations.

To prevent WAR violations, a forwarding mechanism from a producer thread to a consumer thread is provided. When a speculative load misses in L1 cache, the PU sends a request for the corresponding cache line on the bus. PUs that are executing predecessor threads probe their L1 cache to see if they have the requested data. If more than one PU owns the data, the PU executing the most recent predecessor thread is chosen to forward the data. If no candidate is found, the L2 cache will supply the data.

The coherence protocol also provides a RAW violation detection mechanism. All words referred by speculative loads are recorded in the cache directory. When a store is executed, its destination address is broadcasted on the bus. Receiving this address, each PU that executes a thread with higher speculation level checks its corresponding cache entry. Violation is detected if the PU has previously loaded a value from the same location. In this case, the cache controller notifies Thread Controller Unit to flush and restart the execution of the thread.

### 3.3 State Transition

To implement our cache protocol, we prepare eight control bits (figure 4) for each word into the cache directory. The first four bits, Modified, Invalid, Forwarded and Load, are used to identify the state of the corresponding word. A word can be in one of the following seven states: *Modified (M)*,*Shared (S)*, *Modified-Loaded (ML)*, *Shared-Loaded (SL)*, *Modified-Forwarded-Loaded (MFL)*, *Shared-Forwarded-Loaded (SFL)* and *Invalid (I)*. These states are encoded by the four state bits as shown in table 1. Modified bit indicates that a memory word has been modified. Invalid bit indicates the word is invalid. Forwarded bit is set when the word was forwarded from a predecessor thread. When any of the predecessor threads is flushed, memory words with the Forwarded bit set are invalidated. It is because the forwarded data may be incorrect due to misspeculation. Load bit is set when the first access to the memory word is a load, and is used to detect dependency violations.

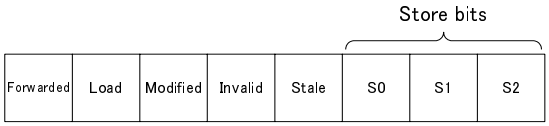


**Fig. 4** State bits maintained for each word in the cache

**Table 1** List of states of a word in the cache

| State | Forwarded | Load | Modified | Invalid |
|---|---|---|---|---|
| M | 0 | 0 | 1 | 0 |
| S | 0 | 0 | 0 | 0 |
| ML | 0 | 1 | 1 | 0 |
| SL | 0 | 1 | 0 | 0 |
| MFL | 1 | 1 | 1 | 0 |
| SFL | 1 | 1 | 0 | 0 |
| I | X | X | X | 1 |

In addition to these four bits, a set of Store bits and a Stale bit are also stored in the cache directory. Store bits identify which of the PUs have stored to the corresponding address. Each Store bit corresponds to one of three PUs, other than the PU the memory word belongs to. Store bits are used to avoid unnecessary violation detections. When a store is executed by another PU, its destination address is broadcasted on the bus. If the store comes from a predecessor thread, cache controller checks both the Load bit and the Store bits. Violation is detected if the Load bit is set, and if none of Store bits corresponding to PUs whose speculation level is higher than the one that broadcasted the store, is set. Stale bit is set at arrival of message indicating that a successor thread has stored into the same memory address. We call this message *Delayed Invalidation message (Dyinv)*. The message indicates that the data can no longer be used by future threads. When a thread commits, all words with Stale bit set are invalidated.

Figure 5 illustrates state transition possible in our cache coherence protocol. List of events as an input to this state diagram is listed in table 2.

## 4. Complexity Analysis

The complexity of our cache model is analyzed in terms of hardware overhead incurred by additional

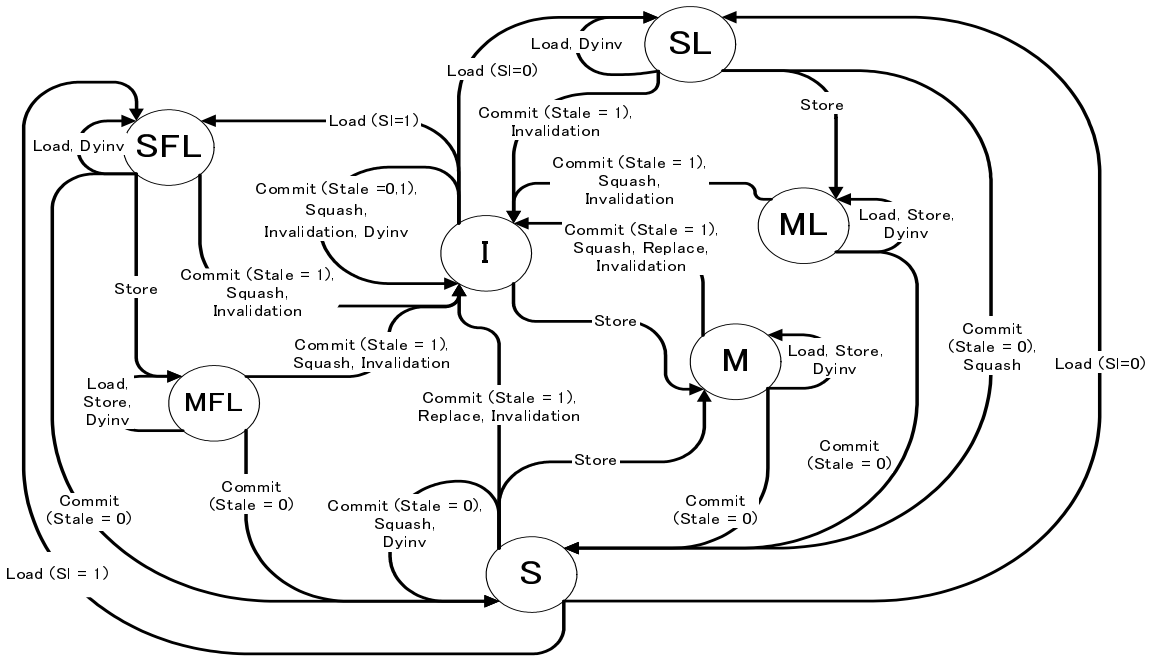


**Fig. 5** State Transition Diagram

**Table 2** List of cache coherence events

| Input | From | Description |
|---|---|---|
| Load (SI = 0) | PU core | The Load issued by the PU and the predecessor thread does not have the data |
| Load (SI = 1) | PU core | The Load issued by the PU and the predecessor thread have the data |
| Store | PU core | The Store issued by the PU |
| Replace | PU core | Cache line replacement |
| Squash | TCU | The thread is squashed |
| Commit (Stale = 0) | TCU | The thread is commit and the Stale bit is set |
| Commit (Stale = 1) | TCU | The thread is commit and the Stale bit isn't set |
| Invalidation | Bus | The store information is broadcasted from a predecessor thread |
| Delayed Invalidation (Dyinv) | Bus | The store information is broadcasted from a successor thread |

memory state bits and control logic. To quantify overhead of the state bits, we estimate cache access time using CACTI tool [7]. Since, the state bits are kept in the cache directory, we compare the access time for the cache directory and for the cache itself, and discuss possible impact on cache access latency.

Delay of additional control logic is estimated using the method of logical effort [10]. Using this method, the delay incurred by a logic gate is calculated as the sum of parasitic delay $p$ and effort delay $f$. The effort delay is further expressed as the product of logical effort $g$, which describes how much bigger than an inverter a gate must be to drive loads as well as the inverter can, and electrical effort $h$, which is the ratio of output to input capacitance of the gate.

$$d = f + p = gh + p \tag{1}$$

Delay along N-stage logic path $D$ is given by the sum of delay through each stage:

$$D = \sum_{i=1}^{N} f_i + \sum_{i=1}^{N} p_i \tag{2}$$

It is known that $D$ is minimum when effort delay through each stage is equal to an optimal effort delay $\hat{f}$:

$$\hat{D} = N\hat{f} + \sum_{i=1}^{N} p_i \tag{3}$$

where $\hat{f}$ is given by

$$\hat{f} = F^{1/N} = (\prod_{i=1}^{N} g_i \prod_{i=1}^{N} b_i \prod_{i=1}^{N} h_i)^{1/N} \tag{4}$$

Here, $b_i$ is branch effort of stage *i*, which estimates the fanout effect of the logic gate in the stage.

To estimate delay overhead of control logic, we model critical path of the logic and calculate $\hat{D}$ along the path. As a measure of the delay, we use the delay through an fanout-of-four (FO4) inverter. It is known that delay normalized by the FO4 metrics holds constant over a wide range of process technologies. To provide a concrete example, absolute delay at 90 nm technology will also be shown, assuming 1 [FO4] = 36 ps.

### 4.1 Cache Directory Access Time

In this section, we discuss overhead incurred by cache state bits. The L1 cache configuration we as-

sume is given in table 3. We assume that cache data, tag, and state bits are each kept in a separate memory structure. Our coherence protocol requires eight state bits per word. Assuming 32-kB cache with 64-byte lines, this results in 4 kB of cache directory in total. Because additional state bits are required to maintain speculative state and the states are kept on a per word basis, the directory size is much larger than the one for conventional coherence protocol. For example, the simplest MSI protocol only needs three state bits per line. Then the directory would be only 192 bytes in size.

**Table 3** L1 cache parameters

| | |
|---|---|
| Size | 32 kB |
| Line Size | 64 bytes |
| Associativity | 2 |
| Tag | 21 bits |

Table 4 shows cache access latency estimated using CACTI. It can be seen the cache directory access latency is less than tag access latency. In this case, the critical path is in tag access operation. Since the directory access can be performed in parallel to tag access, it is expected that the directory would not affect cache access latency. In this estimation, we assume 32 bit address space. In larger address space, tag array would be larger and require more access time, so that cache directory is less likely to come on the critical path.

**Table 4** Cache access time

| | Delay (FO4) | Delay @90nm [ps] |
|---|---|---|
| Data | 17.3 | 622 |
| Tag | 19.1 | 689 |
| Directory | 11.4 | 410 |

## 4.2 Control Logic Delay

We estimate complexity of control logic required for four main operations of the cache controller: *owner probing* and *data transfer* for data forwarding between PU, *violation detection* and *state transition*. Critical paths of each logic block are illustrated in figure 6 $\sim$ 9, along with their estimated delay. Functions of those logics are described briefly below.

### Forward - Owner Probing

On receiving a forward request, each PU checks if it has the requested line in the cache. Then it checks modified bit for each word in the line, and if the bit is set, asserts *owner PU line* to claim its ownership.

### Forward - Data Transfer

Each PU examines which PUs have claimed ownership for the requested words. It compares speculation level of all the owner PUs, and checks if it is qualified to forward the word. If it is, it puts the word on the bus, and if not, send a request to L2 cache.

### Violation Detection

On receiving a broadcast that a store has been executed, each PU first identifies which PUs have stored to the corresponding word by checking its store bits. It then compares speculation level of those PUs with its own speculation level and with that of the latest store. Finally it checks modified bit and load bit of the word and detects violation.

### State Transition

Manage the update of state bits in the cache directory as previously described in figure 5 and table 2.

| | Delay [FO4] | Delay @90nm [ps] |
|---|---|---|
| Owner Probing | 13.9 | 500 |
| Data Transfer | 7.3 | 263 |
| Violation Detection | 18.6 | 668 |
| State Transition | 21.2 | 763 |

**Table 5** Operation delay including access time to cache directory

Table 5 summarizes estimated time needed for each operation. The operation times include access time to

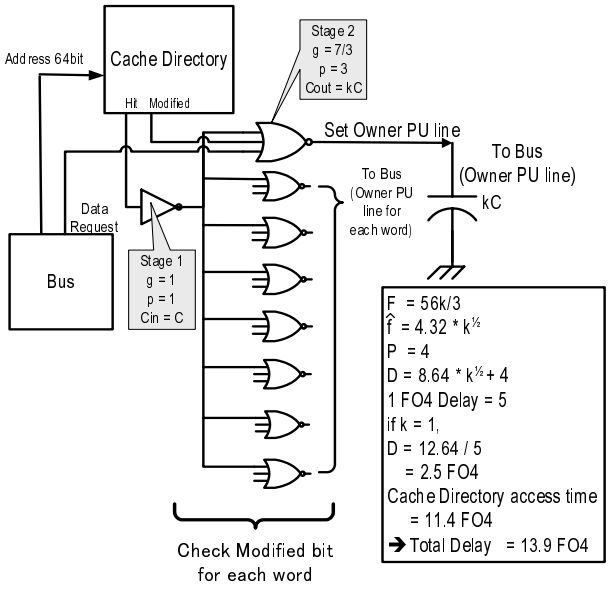


**Fig. 6** Forward - Owner Probing

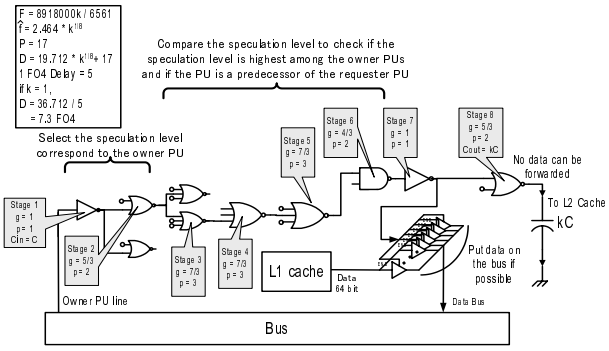


**Fig. 7** Forward - Data Transfer

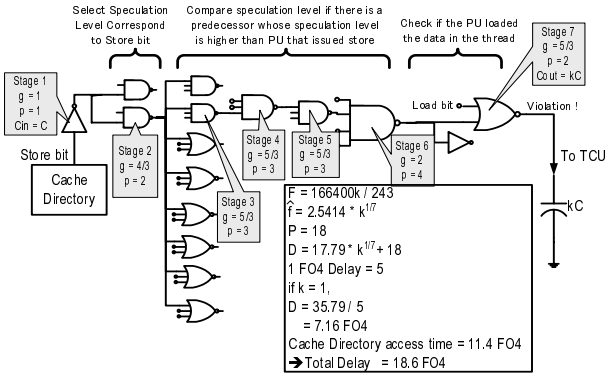


**Fig. 8** Violation Detection

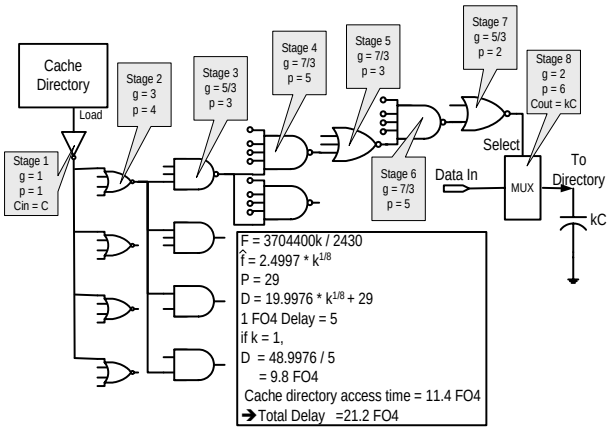


**Fig. 9** State Transition

cache directory, which has been previously estimated to be 11.4 [FO4] (table 4). It can be seen that directory overhead occupies a large part of path delay. The results also indicate that the control logic slightly extend the critical path by 11% from 19.1 [FO4] (tag access latency) to 21.2 [FO4] (state transition update). Overall, it is estimated that the cache controller can operate at a reasonable cycle time of less than 22 [FO4]. This can be made faster by applying optimizations such as pipelining the directory access and logic operation.

## 5. Conclusion

A number of memory speculation mechanisms in speculative multithreading chip multiprocessors have been proposed and many performance studies have been reported. However, it is still doubtful whether the mechanism is complexity effective to implement. In this paper, we performed a complexity analysis of a cache controller designed by extending an MSI controller to support thread-level memory speculation. We modeled and estimated the delay of logic on critical paths and area overhead to hold additional control bits in cache directory.

We found that the increase in area of cache directory over the original MSI cache directory is significant. This is a consequence of the requirements to maintain speculative state of memory on a per-word basis rather than per-line basis. For many protocol operations, area overhead occupied more than half of the total delay. This area overhead is however smaller than the delay for accessing and comparing cache tags. When the protocol operations are performed in parallel with tag comparison, the critical path latency is increased by 11%. Overall, our results showed the cache controller can be implemented with cycle time of less than 22 [FO4]. The cycle time can be made shorter if we further pipeline the protocol operations, for example by splitting cache directory access and logic operation. The impact of this optimization to the performance is the future work of this research.

## Acknowledgement

This research is partially supported by Grant-in-Aid for Fundamental Scientific Research B(2) #13480077 from Ministry of Education, Culture, Sports, Science and Technology Japan, Semiconductor Technology Academic Research Center (STARC) Japan, CREST project of Japan Science and Technology Corporation, and by 21st century COE project of Japan Society for the Promotion of Science.

## References

[1] N. D. Barli, H. Mine, S. Sakai, and H. Tanaka. A Thread Partitioning Algorithm using Structural Analysis. *ARC-2000-139*, 2000(24):37–42, 2000.

[2] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5):552–557, 1996.

[3] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proc. of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, 1998.

[4] 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.

[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] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical Report WRL-2001-2 HP Labs Technical Reports, 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] I. E. Sutherland, R. F. Sproull, and D. Harris. *Logical Effort: Designing Fast Cmos Circuits*. Morgan Kaufmann, 1999.

[11] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):881–902, 1999.