Vol. 45 No. 7 IPSJ Journal July 2004

Regular Paper

Cache Coherence Strategies for Speculative Multithreading CMPs : Characterization and Performance Study

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

Thread-level memory speculation is one of speculation techniques usually employed in speculative multithreading architectures. On shared-bus chip multiprocessors (CMPs), the technique can be implemented by extending their cache coherence mechanisms. Several implementations have been proposed and evaluated. However, there is no study that compares the impact of the taken strategies and identifies which of the strategies are important. In this paper, we first characterize the effect of speculative multithreading executions to cache misses. We find that sharing misses occupy the largest portion of misses, and spreading accesses in speculative multithreading executions cause a significant increase in the miss rate. Then, we perform a performance study of several cache coherence strategies. Our study shows that update-based protocols, which do not suffer from sharing misses, achieve significantly higher performance than invalidation-based protocols. The study also reveals that applying a modified read-broadcast (snarfing) approach is effective for suppressing the effect of spreading cache accesses. Finally, the study shows that managing the exclusivity of data in caches offers little performance improvement.

1. Introduction

Speculative multithreading has been proposed to accelerate the execution of sequential programs by dividing a sequential execution stream into threads and executing the threads speculatively[1),4),8),14)~16),19),21),23)]. In contrast to conventional parallelization approaches, control and data dependencies may exist among the threads. To guarantee the correctness of the execution, a number of hardware and software supports are typically integrated to the underlying architectures.

Thread-level memory speculation is one of speculation techniques usually employed in speculative multithreading architectures. A speculative thread may issue memory operations without waiting for memory operations of its predecessor threads to complete. This implies that we need to buffer values stored by the speculative threads, perform version management of memory values, and detect dependency violations. These functions can be implemented on shared-bus chip multiprocessors (CMPs) by extending their cache coherence mechanisms. Several implementations have been proposed and evaluated[12)~14),20)]. However, to our extent of knowledge, there is no study that compares the impact of the taken strategies and identifies which of the strategies are important.

In this paper, we first characterize the effect of speculative multithreading executions to cache misses. We find that sharing misses occupy the largest portion of misses, and spreading cache accesses in speculative multithreading executions causes a significant increase in the miss rate. These results suggest that eliminating the sharing misses and suppressing the effect of spreading accesses are important factors for achieving good performance.

We then perform a performance study of several cache coherence strategies. Our study shows that update-based protocols, which do not suffer from sharing misses, achieve significantly higher performance than invalidation-based protocols. Update-based protocols also have lower bus utilization. These results are interesting since they contradict previous results in the context of conventional multithreading executions on multiprocessors. In this context, update-based protocols often suffer from a large increase in bus utilization, outweighing the benefit of lower cache miss rate[7),9),10)].

The study also reveals that applying a modified read-broadcast (snarfing) approach is effec-

† Graduate School of Information Science and Technology, The University of Tokyo
☆ Presently with Texas Instruments Japan
☆☆ Presently with East Japan Railway Company
☆☆☆ Presently with Institute of Information Security

tive for reducing the effect of spreading cache accesses. Originally, read-broadcast approach is intended to reduce sharing misses in multiprocessor caches[11),17)]. The bus is snooped for read-miss transactions, and whenever a response appears on the bus and there is a matching tag of an invalid line, the line is inserted in the cache. The original approach is however not sufficient since the spreading cache accesses also cause a significant increase in non-sharing misses. The modified read-broadcast approach extends the original approach by inserting the snooped line not only when an invalid copy of the line found, but also when there is no matching copy of the line in the cache.

Finally, our study also shows that managing the exclusivity of data in caches offers little performance improvement. This suggests that exclusivity management can be traded off if a simpler design in the cache coherence protocol is preferred.

The rest of this paper is organized as follows. Section 2 presents the related work of cache coherence protocols that support speculative multithreading executions. Section 3 describes the methodology of this study. The results of the study are presented and discussed in Section 4. Finally Section 5 concludes the paper.

2. Related work

Several cache coherence protocols for speculative multithreading CMPs have been proposed[12)~14),20)] and an analysis of the complexity involved has been reported[24)]. Speculative versioning cache (SVC)[12)] proposed for Multiscalar architecture, uses an invalidation-based protocol. The state of data is managed on a per-word basis and a linked-list structure is used for version management. SVC employs a slightly modified read-broadcasted approach in which the line is snarfed if there is a free line available.

Hydra[13)] also uses an invalidation-based protocol. The state of data is managed on a per-line basis. Write-through policy is employed: non-speculative writes are written directly through to the L2 cache, while speculative writes are temporarily stored into dedicated speculative buffers attached to the bus.

STAMPede[20)] extends the MESI invalidation-based protocol to support memory speculation. Similar to Hydra, the state of data is managed on a per-line basis. However, it retains speculative values in the cache.

In IACOMA[14)], memory speculation is supported 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. The state of data is managed on a per-word basis. Similar to STAMPede, speculative values are retained in the L1 caches.

SVC, Hydra, STAMPede, and IACOMA, all use invalidation-based protocols. Our study however reveals that update-based protocols considerably reduce the number of cache misses and offer a significant performance improvement over invalidation-based protocols. The study also shows that, applying the modified read-broadcast approach to the update-based protocols further improves the performance achieved.

3. Methodology

3.1 Execution model

In this paper, we define *thread* as a connected subgraph of the control flow graph with a single entry node. This definition is similar to the one used in Multiscalar[19)]. It differs in that we allow overlap among the tail regions of different threads[2)].

A program is partitioned into threads by a compiler. We incorporated a partitioning algorithm into an improved version of *newcc* optimizing compiler[3)] originally developed by Fujitsu Laboratories. The compiler first finds thread candidates by analyzing the program's control flow graph using structural analysis[18)]. In our algorithm, structural analysis is configured to identify structures that meet the requirements of our thread model described above. These structures include if-then blocks, if-then-else blocks, and loop structures.

After finding the thread candidates, the compiler applies a heuristic approach for selecting a combination of threads that is most likely give optimal performance. Here, we use a simple heuristic that mainly considers loop structures and thread size characteristic. We first identify innermost loop structures from the candidates. Then, we select a combination that includes the loop structures and minimizes the total number of threads. Furthermore, only for candidates whose estimated size exceed a preset threshold value, we then insert a thread header into each control flow edge that enters the candidates. Note that this thread header indicates the start of a thread and by inserting

Table 1 Average dynamic size of threads and the coverage of parallel execution

Application	thread size [insts]	coverage [%]
099.go	25.98	99.69
124.m88ksim	24.30	89.99
126.gcc	26.92	90.12
129.compress	20.66	99.89
130.li	22.18	99.82
132.ijpeg	51.61	94.71
134.perl	29.05	66.23
147.vortex	31.09	80.68
harmonic mean	27.99	88.57

the headers into the edges rather than to the entry blocks of the thread candidates allows us to overlap the tail regions of two or more threads. Overlapping tail regions is useful to avoid the creation of tiny threads that was a problem for sustaining a large execution window[2]. **Table 1** shows the average dynamic size of threads and the coverage of parallel execution in our evaluation environment.

During execution, a thread predictor[5] dynamically predicts the sequence of threads that follow the program order. The threads are then assigned to the processing units of the CMP in a round-robin fashion. If a misprediction occurs, the mispredicted thread and all its successors are flushed and the execution is restarted.

Inter-thread register dependencies are handled using a synchronization mechanism[6]. The compiler helps the hardware to identify which registers to synchronize and the timing to safely send the values. In contrast, inter-thread memory dependencies are handled in a speculative way. A thread may issue memory operations without waiting until its predecessors complete all memory operations. If a predecessor thread afterwards stores to a location previously loaded by the thread, a dependency violation is detected, and the violating thread and all its successors are flushed and restarted.

3.2 Simulation setup

Eight applications from SPEC95INT benchmark are used for the simulations. The parameters, shown in **Table 2**, are set so that one execution will finish in 100 to 300 million instructions.

We use two types of simulator in this paper. The first simulator is a trace-based cache simulator used for characterizing the impact of partitioning a sequential program into threads to cache miss rate. We first generate traces that record memory accesses performed by the threads. The threads are assumed to be perfectly predicted and scheduled to the processing unit in a round-robin fashion. The sequence of the threads follows the program order as defined in our execution model. The memory trace in each thread also follows the program order. Given these traces, the simulator then simulates the behavior of the caches and classifies the occurring cache misses. The timing of cache update is ideal. When a miss occurs, the corresponding line is assumed to be brought to the cache immediately before another access may occur.

Table 2 SPEC95INT benchmark parameters

Application	Parameter
099.go	9 9
124.m88ksim	train/ctl.in
126.gcc	-quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -O genrecog.i genrecog.s
129.compress	30000 1 2131
130.li	train.lsp
132.ijpeg	-image_file specmun.ppm -compression.quality 50 -compression.optimize_coding 0 -compression.smoothing_factor 50 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.findoptcomp
134.perl	scrabble.pl scrabble.in
147.vortex	train/vortex.in

In this simulator, since our purpose is to characterize the sharing misses and the effect of spreading accesses introduced by speculative multithreading executions, we simulate neither the parallel execution of the threads nor the superscalar execution. All memory accesses by predecessor threads must have been issued before a new thread is scheduled to a processing unit. It is because at this evaluation stage, we want to isolate the inherent characteristics from the side effects of the speculative executions (i.e. memory accesses by misspeculated threads or by instructions that follow a mispredicted branch).

For example, assume that a thread stores to a location whose copy also exists in the cache of a processing unit running one of its successors. Also assume that the successor thread loads the value from the location. This example illustrates a producer-consumer sharing pattern that *inherently* causes a sharing miss in the con-

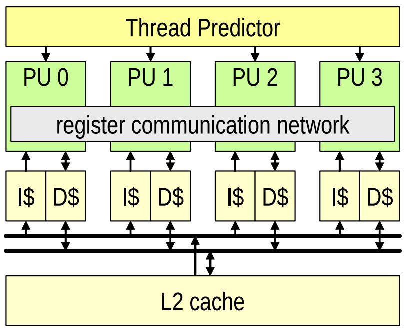

Fig. 1 CMP organization used in this study

Table 3 Processing unit and cache parameters

Parameter	Value
Pipeline stages	7 stages
Fetch/Issue/Retire width	4 instructions/cycle
No. of physical registers	128 registers
Functional units	2 ALUs + 2 Addr. Units
Reorder buffer	64 entries
Issue queue	20 entries
Load/store queue	20 entries
BTB	1024 entries
Bimodal predictor	4096 entries
L1 instruction cache	16-kB 64-B line 2-way set associative 1-cycle hit latency
L1 data cache	16-kB 64-B line 2-way set associative 2-cycle hit latency
L2 unified cache	ideal (always hit) 16-cycle latency

sumer's cache in invalidation-based protocols. However, in real speculative multithreading execution, the consumer may speculatively load the value resulting in a cache hit. It is then followed by a store by the producer, causing a memory dependency violation and invalidation of the line. The consumer thread is restarted and eventually loads from the location, now resulting in a cache miss. In this case, the first cache hit is irrelevant and better not to be included in the statistics taken.

The second simulator is an execution-driven cycle-accurate simulator used for the performance study in this paper. It simulates the aspects of out-of-order superscalar and speculative multithreading executions, including branch prediction, speculative spawning of threads, and speculative accesses to the memory system. The simulator simulates a CMP consisting of four processing units. Each processing unit is an out-of-order superscalar core with a 7-stage pipeline. The processing unit can fetch, issue, and retire four instructions per cycle. Each processing unit has private L1 instruction and data caches, and shares a unified L2 cache. The hardware organization of the CMP is shown in **Fig. 1**. The parameters of the processing units and their caches are summarized in **Table 3**.

Table 4 Types of protocols used in this study

Name	Description
inv	invalidation-based protocol no read-broadcast
inv-robr	invalidation-based protocol read-broadcast in read misses
upd	update-based protocol no read-broadcast
upd-robr	update-based protocol read-broadcast in read misses
upd-rwbr	update-based protocol read-broadcast in both read/write misses

The simulator incorporates a dual-length path-based predictor[5] with lazy update policy as the dynamic thread predictor. We combine two path-based predictors with path length of one and four respectively. Each predictor uses a 2048-entry prediction table. The selection table has 4096 entries of 3-bit counter. The predictor is able to predict one thread in each cycle. For the SPEC95INT benchmark we use in this paper, the prediction accuracy achieved ranges from 62% (*099.go*) to 95% (*124.m88ksim*), with an average of 82%.

For register communication, we assume a register communication mechanism previously proposed in[6]. The update and propagate latency to an adjacent processing unit is set to one cycle, while the update and propagate bandwidth is set to one register/cycle.

For cache coherence protocol of the data caches, the simulator incorporates five types of protocols to be described in the next section. Memory dependency violations are detected on a per-word basis. A memory violation flushes the execution of the current thread and the execution is restarted after one cycle penalty.

Finally, the simulator simulates a shared-bus that connects the data caches. The bus model and its parameters are described in section 3.4. For instruction caches, we do not model bus contention. However, we apply the read-broadcast approach to avoid misses caused by the spreading accesses to the instruction cache.

3.3 Coherence protocols

In this study, we assume a non-blocking write-back write-allocate cache as the base design. Speculatively stored data is retained in

Table 5 Processor and bus events

Name	Description
PrRd	Processor read (load)
PrWr	Processor write (store)
PrNonSpec	The thread becomes non-speculative
PrFlush	The thread is flushed
PrCommit	The thread is committed
PrEvict	The data is evicted from the cache
PrInv	Invalidate the data
PrUpd	Update the data
BusWb	Bus write-back transaction
BusRd	Bus read transaction
BusRdX	Bus read-with-intent-to-modify
BusUpd	Bus update transaction (upd-based)
BusUpg	Bus upgrade transaction (inv-based)
BusFwd	Bus forward (part of BusRd/BusRdX)

Table 6 The basic states of cache data

State		Valid	Excl.	Owned
Invalid	I	-	-	-
Clean Shared	S	√	-	-
Clean Exclusive	E	√	√	-
Modified Shared	O	√	-	√
Modified Exclusive	M	√	√	√

the cache. Cache fill and eviction are performed on a per-line basis while the state of data is managed on a per-word basis.

We consider five types of protocols as shown in **Table 4**. For each protocol, we define two variants: one that manage and one that does not manage the exclusivity of data in the cache. The set of processor and bus events assumed in the protocols are summarized in **Table 5**. Note that bus upgrade transaction (*BusUpg*) is specific to invalidation-based protocols while bus update transaction (*BusUpd*) is specific to update-based protocols.

In conventional cache coherence protocols, there are basically three properties used for defining the state of data in the cache: *Validity*, *Exclusivity*, and *Ownership*[22]. Using these properties, we can define five basic states as shown in **Table 6**. These states are sufficient for describing cache coherence protocols when there is only one version of valid data in the cache. Unfortunately, in speculative multithreading executions, it is possible to have multiple versions of valid data at the same time.

To correctly describe the states of data in speculative multithreading executions, we need to define four additional properties described as follows:

- *Speculativeness*: Data is speculative if it is stored or forwarded by a speculative thread. When a thread is flushed due to misspeculation, all speculative data in the corresponding cache must be invalidated. Speculativeness of data is reset when the thread is promoted to non-speculative thread.
- *Violation possibility*: Data is possible to cause memory dependency violation when the first speculative access to the data is a load. Violation is detected when a less speculative thread later stores to the corresponding location of the data. Violation possibility is reset when the thread is promoted to non-speculative thread.
- *Committed data*: A modification to data is said to be committed when the thread that made the modification retired. The cache may retain the data after the thread retired. However, it is responsible to write the data back before invalidating or replacing the data. For each data, there may be only one cache that has the committed property set. This property has no meaning for non-modified data.
- *Delayed invalidation*: Data is delayed-invalidated if a later (more speculative) thread stored the the same location. A delayed-invalidated data must be invalidated before a new thread is assigned to the corresponding processing unit. If the data is committed, it must be written back before being invalidated.

These new properties increase the number of possible states in speculative multithreading caches. Showing all possible state transitions consumes a lot of space and is difficult to comprehend. We simplify the description as follows. Each property described above can be seen as a dimension in a multi-dimension state space. Thus, we have a seven-dimension state space. The state space can be separated into several sets of independent domains. We can then equivalently describe the protocol by defining the transition rules among the domains in each set. Note that an event may initiate domain transition in more than one sets at a time.

We define five sets of domains as follows:
(1) speculativeness cleared (u) or set (U)
(2) violation possibility cleared (v) or set (V)
(3) committed data cleared (c) or set (C)
(4) delayed invalidation cleared (d) or set (D)
(5) M, O, E, S, or I

The first four sets each have two domains, characterized by whether one of the four properties specific to speculative multithreading executions is cleared or set. The fifth set has five

Table 7 Rules for transition in domains specific to speculative multithreading executions

curr	event	condition	action	next
u	PrWr	thread is speculative	-	U
u	BusRd	forwarded by speculative thread and version matched	-	U
u	BusRdX	forwarded by speculative thread and version matched	-	U
u	BusUpd	updated by speculative thread and version matched	-	U
U	PrNonSpec	-	-	u
U	PrFlush	-	PrInv	u
v	PrRd	first speculative access	-	V
V	PrNonSpec	-	-	v
V	BusRdX	version matched	PrFlush	v
V	BusUpd	version matched	PrFlush	v
V	BusUpg	version matched	PrFlush	v
c	PrCommit	the data is modified	-	C
C	PrEvict	-	BusWb	c
C	BusRdX	version matched	BusWb	c
C	BusUpd	version matched	BusWb	c
C	BusUpg	version matched	BusWb	c
d	BusRdX	by more speculative thread	-	D
d	BusUpd	by more speculative thread	-	D
d	BusUpg	by more speculative thread	-	D
D	PrCommit	-	PrInv	d

Table 8 Rules for transition in MOESI domains

			Exclusivity managed				Exclusivity not managed			
			update		invalidation		update		invalidation	
curr	event	condition	action	next	action	next	action	next	action	next
I	PrRd	shared data	BusRd	S	BusRd	S	BusRd	S	BusRd	S
I	PrRd	unshared data	BusRd	E	BusRd	E	BusRd	S	BusRd	S
I	PrWr	shared data	BusRdX	O	BusRdX	M	BusRdX	O	BusRdX	O
I	PrWr	unshared data	BusRdX	M	BusRdX	M	BusRdX	O	BusRdX	O
{I}	BusRd	-	-	S	-	S	-	S	-	S
{I}	BusRdX	-	-	S	-	I	-	S	-	I
S	PrInv	-	-	I	-	I	-	I	-	I
S	PrWr	-	BusUpd	O	BusUpg	M	BusUpd	O	BusUpg	O
S	BusRdX	-	PrUpd	S	PrInv	I	PrUpd	S	PrInv	I
S	BusUpd	-	PrUpd	S	PrInv	I	PrUpd	S	PrInv	I
E	PrInv	-	-	I	-	I	-	-	-	-
E	PrWr	-	-	M	-	M	-	-	-	-
E	BusRdX	-	PrUpd	S	PrInv	I	-	-	-	-
E	BusUpd	-	PrUpd	S	PrInv	I	-	-	-	-
O	PrInv	-	BusWb	I	BusWb	I	BusWb	I	BusWb	I
O	PrWr	-	BusUpd	O	BusUpg	M	BusUpd	O	BusUpg	O
O	BusRd	-	BusFwd	O	BusFwd	O	BusFwd	O	BusFwd	O
M	PrInv	-	BusWb	I	BusWb	I	-	-	-	-
M	BusRd	-	BusFwd	O	BusFwd	O	-	-	-	-

common condition : the version of the data involved matched

domains characterized by the three properties common to conventional multiprocessor caches.

Table 7 shows the transitions among the domains in the first four sets. The fields in the table are: the current domain, the event that initiates the transition, conditions necessary for the transition, the action taken by the cache controller when the condition is satisfied, and the next domain after the transition. The rules in this table are common to all types of protocols we consider in this paper.

The condition "version matched" indicates whether the version of data involved is relevant to the current thread running on the corresponding processing unit. In other words, when the version is matched, if the thread performs a load from the location, the same version of data as the one under consideration will be provided.

Table 8 shows the transitions among the MOESI domains in the fifth set. The condition "version matched" is applied to all the transitions in this table. We show the rules for update-based and invalidation-based protocols. The fifth and sixth rows in the table show the rules that are relevant only if the read-broadcast capability is enabled (the fifth row

Table 9 Latency settings for each stage in address and data tenure

Trans.	Address tenure	Data tenure
	Arb-Addr-Fin	Ovh-Arb-Ctrl-Data-Fin
BusWb	1 - 1 - 1	0 - 1 - 4 - 4 - 1
BusRd	1 - 1 - 1	6 - 1 - 4 - 4 - 1
BusRdX	1 - 1 - 1	6 - 1 - 4 - 4 - 1
BusUpd	1 - 1 - 1	0 - 1 - 1 - 1 - 1
BusUpg	1 - 1 - 1	0 - 0 - 0 - 0 - 0

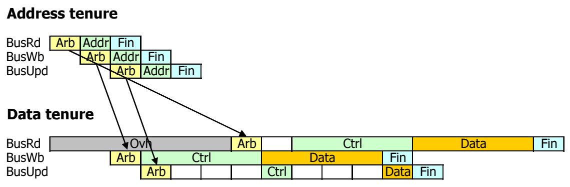

Fig. 2 Example of address and data tenure of the split transaction bus

is for read-broadcast on read misses while the sixth row is for read-broadcast on write misses).

For each class of protocols, we show the two variants that manage and does not manage the exclusivity of data. Note that in the variants that do not manage the exclusivity of data, transitions to E or M domain are not possible. Also note that update-based protocols broadcast cache updates using *BusUpd* or *BusRdX* transactions, while invalidation-based protocols broadcast invalidation messages using *BusUpg* or *BusRdX* transactions. Finally, also note that no update or invalidation is broadcast if the data is exclusive (in E or M domain).

3.4 Bus model

The execution-driven cycle-accurate simulator models a split transaction bus with out-of-order completion. A transaction consists of an address tenure and a data tenure. Address tenure is pipelined into three stages: Arbitration (*Arb*), Address (*Addr*), and Final (*Fin*). The type and memory address of the transaction are broadcast at *Addr* stage. The corresponding data tenure can be started the next cycle after the *Arb* stage of the address tenure.

Data tenure is pipelined into five stages: Overhead (*Ovh*), Arbitration (*Arb*), Control (*Ctrl*), Data (*Data*), and Final (*Fin*). *Ovh* stage is used to simulate latency necessary for accessing lower level (L2) cache array and is fully pipelined. *Ctrl* stage is used to simulate latency necessary for version identification. *Data* stage is where the corresponding data is broadcast on the bus.

For address bus arbitration, we applied an arbitration algorithm that prioritizes requests by a less speculative thread over requests by a more speculative thread. A transaction that wins the arbitration is inserted into a waiting queue. Some transactions require several cycles of overhead before becoming ready for data bus arbitration. Transactions are issued for data bus arbitration from the queue in an out-of-order fashion. When there are more than one ready transactions, the oldest one wins the arbitration.

The length of *Ovh*, *Ctrl*, and *Data* stages depends on the type of the transaction. **Table 9** summarizes the latency setting we use in the simulations. Note that *BusUpg* transaction does not have data tenure, and *BusUpd* transaction occupies fewer cycles of the data bus since it only needs to transfer one word rather than the whole cache line. We assume a data bus width of 16 bytes, so that it takes four cycles to transfer a 64-byte cache line. Also note that *BusRd* and *BusRdX* have a six cycle overhead before the data arbitration can be started.

Figure 2 illustrates an example of the operation of the split transaction bus. It shows three bus transactions, *BusRd*, *BusWb*, and *BusUpd* scheduled to the bus. After competing for the address bus, they are scheduled to the pipeline for data tenure. Since *BusRd* involves additional overhead before being able to arbitrate for data bus, the later two transactions are completed first.

4. Results

4.1 Classification of misses

Figure 3 shows the classification of data cache misses in *inv* protocol. We assume executions on a CMP with four processing units, each with a private 16-kB 64B-line 2-way set-associative data cache. We classify the cache misses into three classes as follows:

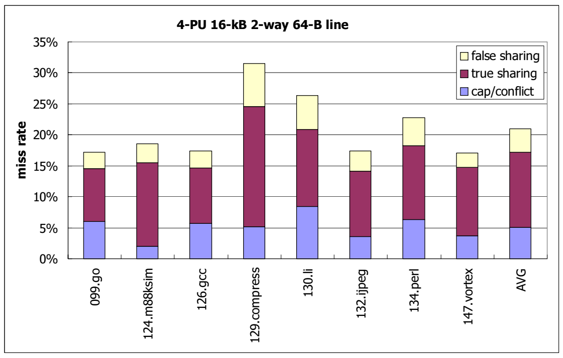

Fig. 3 Data cache miss classification in *inv* protocol

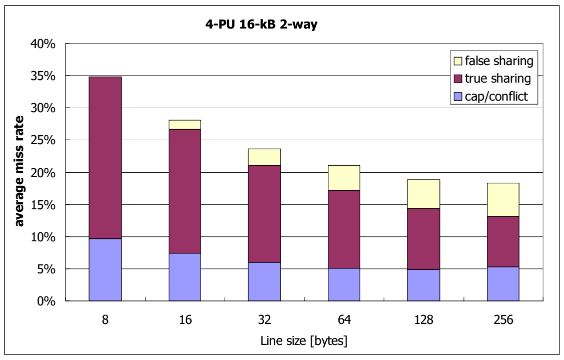

Fig. 4 Varying the line size and its impact to cache miss rate

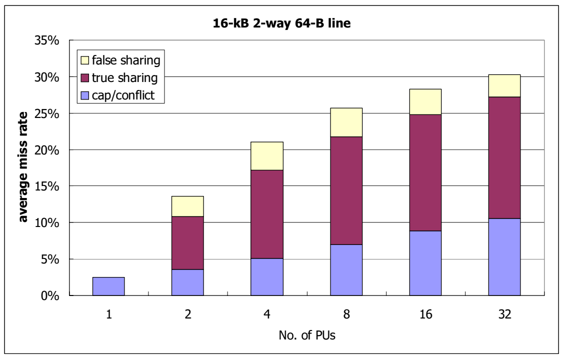

Fig. 5 Varying the number of processing units and its impact to cache miss rate

- *Capacity/conflict misses*: occur when the cache cannot contain all the lines, or when too many lines map to the same cache set. We also include cold misses in this category. However, in most cases, the portion of cold misses is small and negligible.
- *True sharing misses*: occur when the corresponding cache line was invalidated by a write to the same word being accessed.
- *False sharing misses*: occur when the corresponding cache line was invalidated by a write to a different word in the same line.

We prioritize true sharing misses to false sharing misses. Even if the first invalidation was to a different word in the line, but if a later invalidation to the line was to the same word that causes the miss, we classify the miss as a true sharing miss. This is because inherently, the miss still occurs even if we manage the validity of data on a per-word basis in which no false sharing miss may occur. The figure shows that sharing misses occupy more than 75% of the total misses. These results imply that a large part of the cache accesses are in the form of producer-consumer sharing. This fact is natural, considering that, the threads in speculative multithreading are the results of partitioning a sequential program.

Figure 4 shows the average miss rate when we vary the line size from 8 bytes to 256 bytes. As we increase the line size, true sharing misses decrease while false sharing misses increase. False sharing misses can be avoided by managing the states on a per-word basis rather than on a per-line basis. This way, if a predecessor thread write to a word in the cache, we only need to invalidate the word written rather than the whole cache line. For a 64-byte line shown in Fig. 4, we can expect that the average miss rate can be reduced to approximately 17%. However, the remaining true sharing misses still occupy more than half of the total misses. These results hint that, using update-based protocols which do not suffer from sharing misses may be a better choice if they do not generate excessive bus traffic.

4.2 Effect of spreading accesses

Another possible problem in implementing coherence protocols for speculative multithreading CMPs is the spreading of the cache accesses over the multiple data caches. The spreading accesses decrease both the spatial locality and temporal locality of memory accesses. To quantify this effect, we vary the number of processing units and measure how the cache miss rate varies. The results, shown in **Fig. 5**, indicate that the miss rate increase significantly confirming the large impact of the spreading accesses.

Looking into each miss category individually, the impact of spreading accesses to capacity/conflict miss rate is larger than to sharing miss rate. Even if the sharing misses can be completely eliminated in update-based protocols, the increase in capacity/conflict miss rate remains as a problem.

A possible approach to improve the situation is by using a modified *read-broadcast* approach. Originally, read-broadcast approach is intended to reduce sharing misses in multiprocessor caches[11),17)]. The bus is snooped for read-miss transactions, and whenever a response appears on the bus and there is a matching tag of an invalid line, the line is inserted in the cache. The original approach is however not sufficient since the spreading accesses also cause a significant increase in non-sharing misses, as shown in Fig. 5. The modified read-broadcast

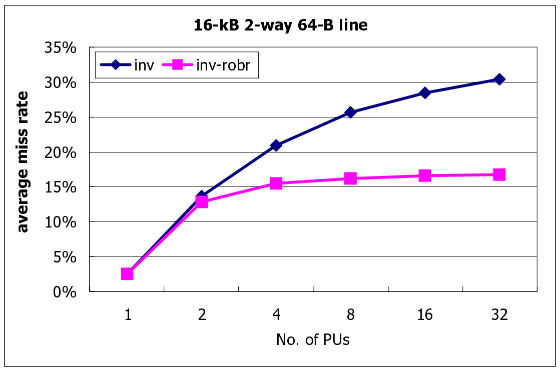

Fig. 6 Estimated read-broadcast effect in invalidation-based protocols

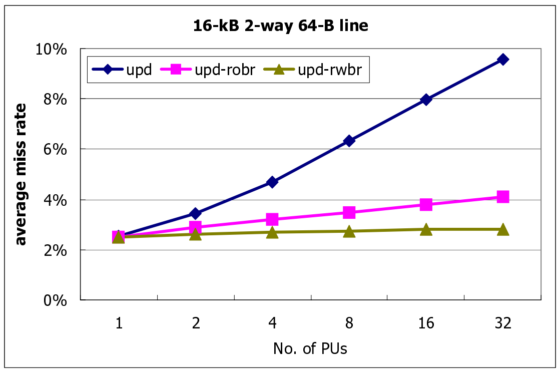

Fig. 7 Estimated read-broadcast effect in update-based protocols

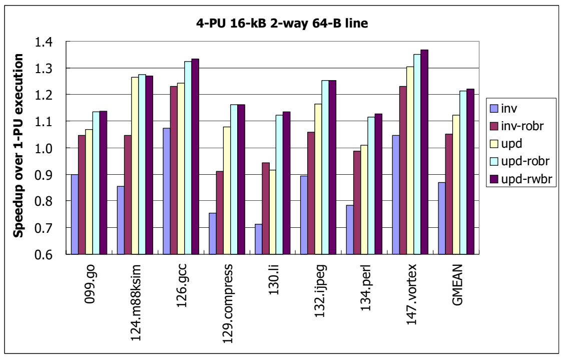

Fig. 8 Speedup achieved using the different types of protocols

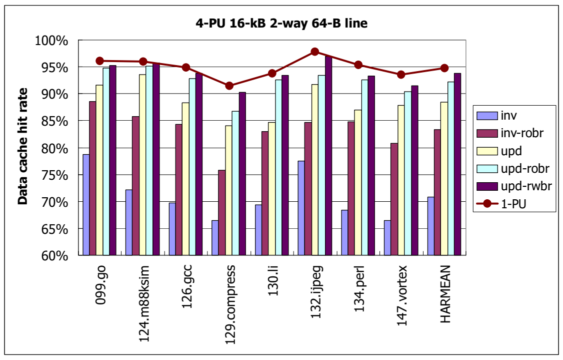

Fig. 9 Data cache hit rate for each type of protocols

approach extends the original approach by inserting the snooped line not only when an invalid copy of the line found in the case, but also when there is no matching copy of the line in the cache. Note that the approach we consider here is more aggressive than the approach taken in Multiscalar's SVC. SVC only inserts the line if there is a free line available.

The logic of the approach is as follows. We expect that there is spatial and temporal locality of memory accesses between a thread and its nearby successors. Thus, when the thread misses a cache line, there is large possibility that the successors may also need the data from the same line. By using the read-broadcast approach to prefetch the line, we can avoid the corresponding misses in the successors. Note that for invalidation-based protocols, the prefetching can be performed only on read miss transactions. On write miss transactions, there is no advantage of prefetching the line since it will be invalidated immediately.

We estimate how much we can suppress the increase in cache miss rate by using the technique. **Figure 6** shows the results for invalidation-based protocols. Comparing the miss rate of *inv-robr* (read-broadcast enabled) to the miss rate of *inv* confirm that we can suppress the increase in cache miss rate using the technique. However, the miss rate in multiple-PU configurations remains significantly higher than the miss rate in single-PU configuration. This is mostly because there are still a lot of sharing misses remaining.

Figure 7 shows the improvement in the case of update-based protocols. From the figure, it can be seen that applying read-broadcast on read misses alone (*upd-robr*) significantly suppresses the increase in cache miss rate. Applying read-broadcast also on write misses (*upd-rwbr*) further reduces the miss rate, helping to maintain the cache miss rate almost constant even when the number of processing units is increased.

4.3 Performance comparisons

Figure 8 shows the speedup over conventional execution on one processing unit

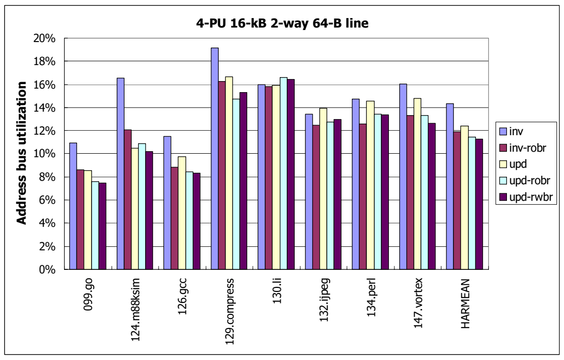

Fig. 10 Address bus utilization

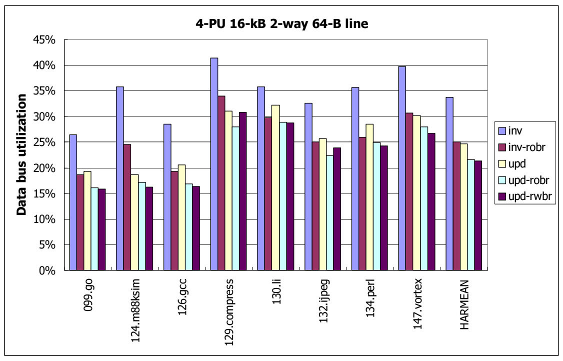

Fig. 11 Data bus utilization

achieved for the five protocols shown in Table 4. Here, we used protocol variants that manage the exclusivity of data. The figure shows that on average the three update-based protocols outperform the other two invalidation-based protocols. The figure also shows that applying read-broadcast approach significantly improves the performance for both invalidation-based and update-based protocols. The impact is however larger for invalidation-based protocols since the cache hit rate in the baseline configuration (read-broadcast disabled) is lower.

We can also observe that applying read-broadcast on both read and write misses as in *upd-rwbr* only slightly improves the performance over *upd-robr*. However, we expect that the impact becomes larger for larger number of processing units as we previously indicated in Fig. 7.

Figure 9 shows the data cache hit rate for each type of protocols. Note that the hit rate is lower than the predicted miss rate shown in Fig. 6 and Fig. 7 since we treat a partial hit (a hit to a cache line being brought to the cache) as a miss. For reference, we also shows the hit rate for conventional execution on a single processing unit. Overall, the results confirm that using update-based protocols instead of invalidation-based protocols, and applying read-broadcast approach significantly reduce the cache miss rate.

4.4 Bus utilization

Figure 10 and **Fig. 11** show the bus utilization of address bus and data bus respectively. We measure the utilization by counting the bus cycles consumed in *Addr* stage for address bus, and in *Data* stage for data bus. The figures show that using update-based protocols and applying read-broadcast approach decrease the bus utilization. The results for update-based protocols are quite surprising since the use of update-based approach works in favor of lower bus utilization.

Figure 12 and **Fig. 13** show the breakdown of the address and data bus cycles consumed in each protocol. They show that, using update-based protocols and applying read-broadcast approach significantly reduce *BusRd* and *BusRdX* transactions caused by cache misses.

Since we set the latency of *Addr* stage to one cycle for all types of transactions, the data shown in Figure 12 can also be interpreted as the breakdown of the number of transactions that occupied the address bus. The figure shows that the number of *BusUpd* transactions in update protocols is at most 80% larger compared to the number of *BusUpg* (invalidation) transactions in invalidation-based protocols. This indicates that there is only a small number of useless update transactions. Furthermore, although the *BusUpd* transactions require additional data bus traffic in update-based protocols, the increase in traffic is offset by the significant decrease in *BusRd* and *BusRdX* traffic.

Finally, we can also observe that applying the modified read-broadcast approach slightly increases the number of bus write-back (*BusWb*) transactions. This is because the prefetching effect of the approach initiates more cache replacements. For most cases, however, the increase is smaller than the reduction in the number of miss transactions (*BusRd* and *BusRdX*), resulting in saved bus cycles.

The reasons why speculative multithreading favors update-based protocols in terms of bus utilization, contradicting previous results in the context of conventional multiprocessors, are as follows:

(1) Update-based protocols reduce sharing

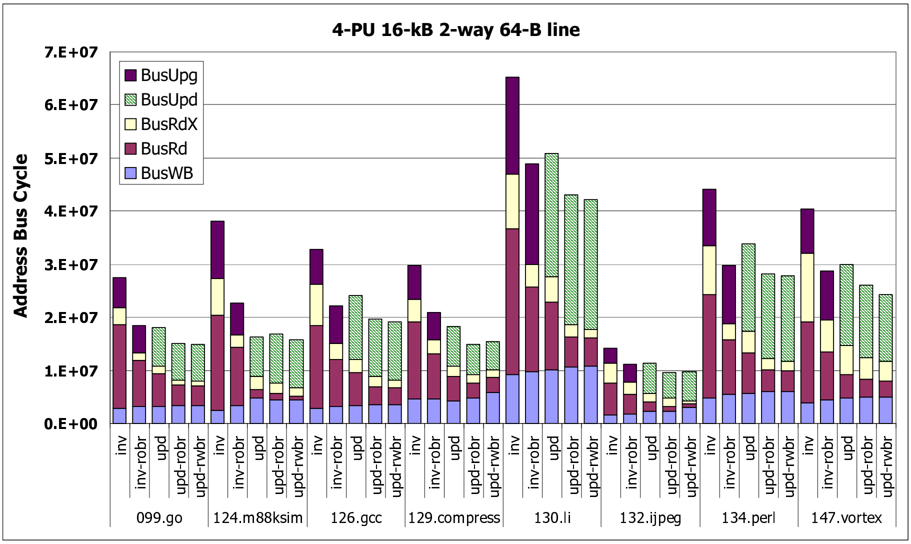

Fig. 12 Address bus cycle breakdown

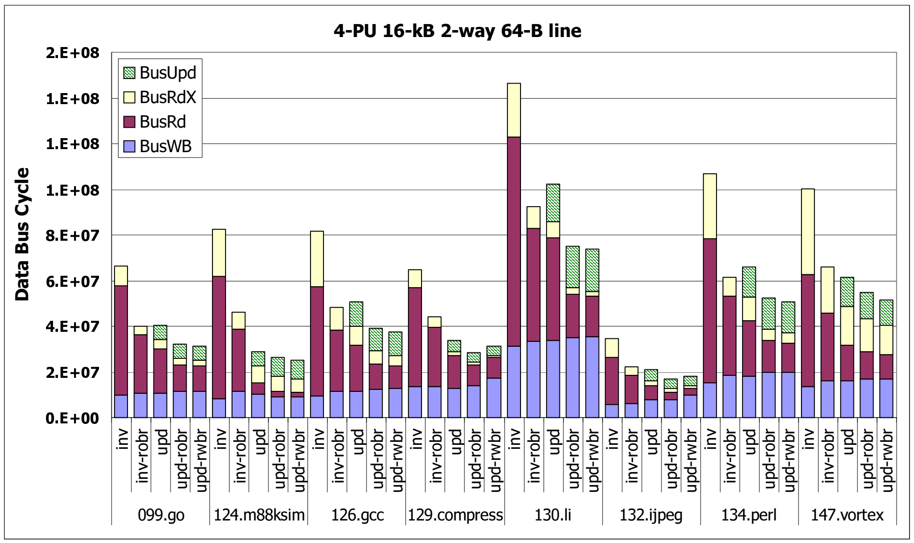

Fig. 13 Data bus cycle breakdown

misses more significantly in speculative multithreading context than in conventional multiprocessor context.

(2) The length of write-runs in speculative multithreading is shorter than in typical workloads of conventional multiprocessors.

Write-run is defined by Eggers et.al. as a sequence of write references to a shared cache data by a single processor uninterrupted by any memory access to the same block by another processor[9]. It is useful as an indicator of whether an application is suitable for invalidation-based protocols or update-based protocols. Application with long write-runs benefits from invalidation-based protocols since it avoids unnecessary updates, while the reverse is true for update-based protocols.

In speculative multithreading, we divide the program into small chunks (threads), thus, there are only a few stores inside a thread. These threads are then assigned to multiple processing units, forming sharing patterns with short write-runs. Invalidation-based protocols perform better if write-run length is considerably large. For example Dahlgren reported that Cholesky has more than 50% of the write-runs with length of eight[7]. In our case, however,

Table 10 Impact of managing exclusivity

	Reduction rate [%]				
Reduction in	inv	inv-robr	upd	upd-robr	upd-rwbr
No. BusUpd	-	-	20.8	26.6	28.7
No. BusUpg	7.9	33.3	-	-	-
Addr. bus cycle	1.9	14.1	12.3	15.5	17.5
Data. bus cycle	-0.2	1.4	7.5	8.0	8.8
Addr. bus utilization	0.3	2.1	1.9	2.2	2.6
Data. bus utilization	-0.1	0.4	1.9	1.8	2.1
Execution cycles	0.2	-0.3	1.1	0.5	0.4

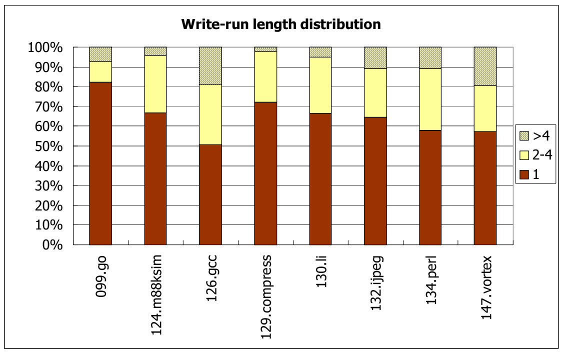

Fig. 14 Write-run length distribution

more than 80% of the write-runs have length less than or equal to four as shown in **Fig. 14**. This, combined with the significant reduction in miss rate, explain why update-based protocols perform better.

4.5 Impact of managing exclusivity

Table 10 presents the impact of managing exclusivity in the five protocols we used in this study. The table shows the reduction in the number of *BusUpd* and *BusUpg* transactions, consumed bus cycles, bus utilization, and the execution cycles. It can be seen that managing exclusivity helps to reduce the number of *BusUpd* and *BusUpg* transactions between 7.9% to 33.3%. These reductions in turn reduce the total number of bus cycles consumed. The reductions in update-based protocols are larger than in invalidation-based protocols, ranging from 12.3% to 17.5% for address bus, and from 7.5% to 8.8% for data bus. These reductions, however, only contribute to a few percents of decrease in bus utilization of the address bus and the data bus. As the result, managing exclusivity only provides a slight performance improvement over the protocols that do not manage the exclusivity. These results suggest that if a simpler design is preferred, it is better to choose protocols that do not manage exclusivity of data.

5. Conclusion

This paper studied the impact of several cache coherence strategies for speculative multithreading chip multiprocessors. We first characterized the effect of speculative multithreading executions to cache misses. We found that sharing misses occupy the largest portion of misses, and spreading accesses in speculative multithreading executions cause a significant increase in the miss rate. These two factors, if not handled properly, limit the speedup that can be achieved using the speculative multithreading executions.

Performance study further showed that update-based protocols, which do not suffer from sharing misses, achieve significantly higher performance than invalidation-based protocols. Interestingly, the update-based protocols also have lower bus utilization. The study also revealed that applying a modified read-broadcast (snarfing) approach is effective for suppressing the effect of spreading cache accesses. Finally, the study showed that managing the exclusivity of data in caches offer little performance improvement, suggesting that it can be traded off if a simpler design is preferred.

Acknowledgments 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. We are also thankful for anonymous reviewers for their constructive critics and suggestions.

References

1) 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. In

Proc. of the JSPP 1998, pages 87–94, 1998.

2) 田代 大輔、バルリ ニコ デムス、坂井 修一、田中 英彦. スレッド投機実行におけるエッジに着目したスレッド分割手法. *ARC-2003-153*, 2003(40):67–72, 2003.

3) 飯塚 大介、小沢 年弘、坂井 修一、田中 英彦. C コンパイラによるループ最適化の検討. *HPC-99-77*, 99(66):65–70, 1999.

4) H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proc. of the 31st MICRO*, pages 226–236, 1998.

5) N.D. Barli, L.D. Hung, H. Miura, S. Sakai, and H. Tanaka. A Dual-Length Path-Based Predictor for Thread Prediction. *International Workshop on Innovative Architectures 2003*, 2003.

6) N.D. Barli, D. Tashiro, C. Iwama, S. Sakai, and H. Tanaka. A Register Communication Mechanism for Speculative Multithreading Chip Multiprocessors. In *Proc. of the SACSIS 2003*, pages 275–282, 2003.

7) F. Dahlgren. Boosting the Performance of Hybrid Snooping Cache Protocols. In *Proc. of the 22th ISCA*, pages 60–69, 1995.

8) M. Edahiro, S. Matsushita, M. Yamashina, and N. Nishi. A Single-Chip Multiprocessor for Smart Terminals. *IEEE Micro*, 20(4):12–20, 2000.

9) S. J. Eggers and R. H. Katz. A Characterization of Sharing in Parallel Programs. In *Proc. of the 15th ISCA*, pages 373–382, 1988.

10) S. J. Eggers and R. H. Katz. Evaluating The Performance of Four Snooping Cache Coherency Protocols. In *Proc. of the 16th ISCA*, pages 2–15, 1989.

11) J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proc. of the 15th ISCA*, pages 422–431, 1988.

12) S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proc. of the 4th HPCA*, pages 195–205, 1998.

13) L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proc. of the 8th ASPLOS*, pages 58–69, 1998.

14) V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

15) P. Marcuello, A. Gonzalez, and J. Tubella. Speculative Multithreaded Processors. In *Proc. of the 12th ICS*, pages 77–84, 1998.

16) K. Olukotun, L. Hammond, and M. Willey. Improving The Performance of Speculatively Parallel Applications on the Hydra CMP. In *Proc. of the 13th ICS*, pages 21–30, 1999.

17) Z. Segall and L. Rudolph. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proc. of the 11th ISCA*, pages 340–347, 1984.

18) M. Sharir. Structural Analaysis: A New Approach to Flow Analysis in Optimizing Compilers. *Computer Languages*, 5(3/4):141–153, 1980.

19) G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd ISCA*, pages 414–425, 1995.

20) J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proc. of the 27th ISCA*, pages 1–12, 2000.

21) J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry. Improving Value Communication for Thread-Level Speculation. In *Proc. of the 8th HPCA*, pages 65–75, 2002.

22) P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proc. of the 13th ISCA*, pages 414–423, 1986.

23) 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.

24) Y. Yanagawa, L. D. Hung, C. Iwama, N. D. Barli, S. Sakai, and H. Tanaka. Complexity Analysis of A Cache Controller for Speculative Multithreading Chip Multiprocessors. In *Proc. of the 10th HiPC (LNCS 2913)*, pages 393–404, 2003.

(Received January 30, 2004)
(Accepted May 9, 2004)

Niko Demus Barli received the M.E. degree in Information Engineering from The University of Tokyo in 2001. He graduated with the Ph.D. degree in Information and Communication Engineering from The University of Tokyo in 2004. His graduate research mainly focused on speculative multithreading techniques on Chip Multiprocessors. He currently works for Texas Instruments Japan.

Luong Dinh Hung is currently a Ph.D. student in Information and Communication Engineering in The University of Tokyo. He received the M.E. degree in Information and Communication Engineering from The University of Tokyo in 2004. He actively pursues new ideas in the field of architecture and circuit techniques for VLSI power reduction.

Hideyuki Miura received the M.E. degree in Information and Communication Engineering from The University of Tokyo in 2004. He currently works for East Japan Railway Company. He investigated and evaluated control speculation techniques and instruction fetch mechanisms for speculative multithreading architectures in his graduate research.

Chitaka Iwama received the M.E. degree in Information and Communication Engineering from The University of Tokyo in 2003. She is currently a Ph.D. student in Information and Communication Engineering in The University of Tokyo. Her research interests are in architecture level power modeling framework and power reduction techniques.

Daisuke Tashiro is currently a Ph.D. candidate in Information and Communication Engineering in The University of Tokyo. He received the M.E. degree in Information Engineering from The University of Tokyo in 2002. His research interests are in speculation and compiler techniques for speculative multithreading architectures.

Shuichi Sakai received the M.E. degree and Ph.D. degree in Information Engineering from The University of Tokyo in 1983 and 1986 respectively. He worked in Electrotechnical Laboratory Japan from 1986 to 1990. From 1991 to 1992, he became a visiting scientist in Computation Structures Group, Massachusetts Institute of Technology. From 1993 to 1996, he was a chief at Massively Parallel Architecture Laboratory in Real World Computing Partnership. He became an Associate Professor in University of Tsukuba in 1996 and came to The University of Tokyo in 1998 as an Associate Professor. From 2001, he has been a Professor in Graduate School of Information Science and Technology of The University of Tokyo. His research interests include dependable computer systems, microprocessor architecture, compiler, parallel computing, and multimedia processing. He wrote several books on logic circuits and computer architecture, including "Introduction to Logic Circuits" and "Computer Architecture with Illustrated Explanation". He is a member of IPSJ, IEEE, ACM, IEICE, and JSAI.

Hidehiko Tanaka received the M.E. degree and Ph.D. degree in Electronis Engineering from The University of Tokyo in 1967 and 1970 respectively. He became an Associate Professor in Graduate School of Engineering of The University of Tokyo in 1971. From 1978 to 1979, he was a visiting professor in The City University of New York. He became a Professor in Graduate School of Engineering of The University of Tokyo in 1987. From 2001 to 2004, he served as the Dean of Graduate School of Information Science and Technology of The University of Tokyo. Currently, he is the Dean of Graduate School of Information Security of Institute of Information Security Japan. His research interests include computer architecture, parallel and distributed processing, natural language processing, multimedia processing, and computer aided design. He wrote many books on computer systems, including “Non Von Neumann Computers”, “Information and Communication Systems”, “Parallel Inference Engine - PIE”, “Computer Architecture”, “VLSI Computer I, II”, and “Software-oriented Architectures”. He is a member of IPSJ, IEEE, ACM, IEICE, JSAI, and JSSST.