A Mechanism for Speculative Memory Accesses Following Synchronizing Operations

Takayuki Sato* Kazuhiko Ohno Hiroshi Nakashima
Toyohashi University of Technology
{taka,ohno,nakasima}@para.tutics.tut.ac.jp

Abstract

*In order to reduce the overhead of synchronizing operations of shared memory multiprocessors, this paper proposes a mechanism, named **specMEM**, to execute memory accesses following a synchronizing operation speculatively before the completion of the synchronization is confirmed. A unique feature of our mechanism is that the detection of speculation failure and the restoration of computational state on the failure are implemented by a small extension of coherent cache. It is also remarkable that operations for speculation on its success and failure are performed in a constant time for each independent of the number of speculative accesses. This is realized by implementing a part of cache tag for cache line state with a simple functional memory.*

This paper also describes an evaluation result of specMEM applied to barrier synchronization. Performance data was obtained by simulation running benchmark programs in SPLASH-2. We found that the execution time of LU decomposition, in which the length of period between a pair of barriers significantly varies because of the fluctuation of computational load, is improved by 13 %.

1 Introduction

A shared memory multiprocessor gives programmers a convenient means for inter-processor communication, which is, of course, its shared memory mechanism. This achieves fine-grain high-speed data transfer in terms of both software owing to small (or often no) overhead of load/store operations, and hardware by means of coherent cache and other mechanisms for access latency reduction and/or hiding.

A communication, however, cannot be fulfilled solely by load/store operations, but has to involve a set of special operations for *synchronization*. A synchronization aims to give a (partial) ordering among ordinary memory accesses performed by communicating processors in order to satisfy true- and/or anti-dependency constraints of the accesses to data to be transferred. Thus a synchronizing operation, for example a lock acquisition, inhibits the successive accesses to memory locations for the communication until the completion of the synchronization is confirmed. The inhibition is often widely applied, for example to all the accesses, for the sake of implementation simplicity.

Since synchronizing operations are usually more costly than ordinary accesses, reducing and/or hiding their costs is an important issue of shared memory architecture and programming. A natural approach is to make communication granularity coarse in order to reduce the frequency of synchronization. This is similar to the technique for distributed memory machines to make messages as large as possible and thus, unfortunately, often mismatches the fine-grain feature of shared memory communication.

For example, in the communication between the processors P_1 and P_2 through the shared variable X_1 shown in Figure 1, the true-dependency constraint (solid arrow) of the read from X_1 by P_2 is obviously satisfied when it arrives to the first barrier at $B_2^a(1)$ because it follows the arrival of P_1 at $B_1^a(1)$. Thus the time from the arrival $B_2^a(1)$ to the departure $B_2^d(1)$ (dark shadow region) spent by P_2 for the confirmation of the synchronization may be unnecessary. Similarly, although X_2 is premature at the barrier arrival of P_1 on $B_1^a(1)$, the true-dependency constraint of the read of X_2 by P_1 is satisfied at a certain point prior to the departure time $B_1^d(1)$. Thus a part of (or the whole of) idle time spent in P_1 may be unnecessary too. Similar observation will be taken for the anti-dependency constraints of X_1 and X_2 (dashed arrow) that the second barrier assures satisfied.

*Currently with Sony Corporation.

†A part of this work is supported by Parallel and Distributed Processing Consortium.

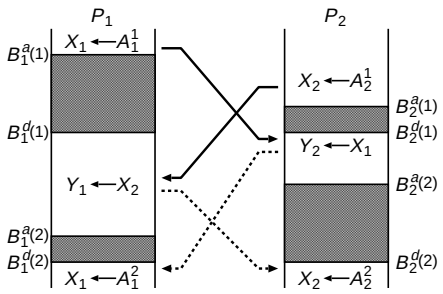

Figure 1. Satisfying Data Dependency Constraint by Barrier Synchronization

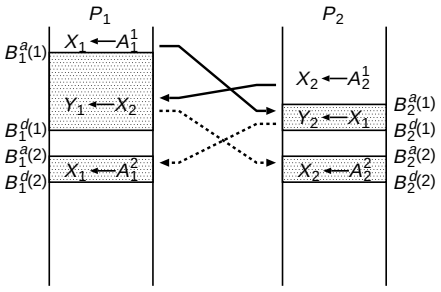

Figure 2. Successful Speculation on Shared Memory Accesses with Barriers

These unnecessary idle times are due to coarsening the communication granularity in which number of true- and anti-dependencies of shared variables are replaced with one control dependency constraint satisfied by a barrier synchronization. Thus we observed that the technique of speculative execution, generally applied to remove/reduce delays caused by control dependencies in uniprocessor execution[20], will be applicable to the idle time reduction.

The rest of this paper describes our speculative memory access mechanism, named *specMEM*, in which an access following a synchronizing operation is performed prior to the confirmation of synchronization assuming the satisfaction of data dependency constraint. In Section 2, we outline how our mechanism works in the case of successful speculation and of failure. Section 3 gives its feasible and efficient implementation model with a small extension of coherent cache. A preliminary experiment result using programs in SPLASH-2 is shown in Section 4. A brief discussion on related works is in Section 5. Finally we conclude the paper summarizing future works in Section 6.

2 Speculative Memory Access: Its Success and Failure

2.1 Successful Speculation and Performance Improvement

In our speculative access mechanism specMEM, a synchronizing operation does not make the processor stalled but turns its execution mode into *speculative* until the completion of the synchronization is confirmed. Therefore, all the memory accesses including those to shared variables are performed as usual assuming that data-dependency constraints, which the synchronizing operation assures satisfied, have already been satisfied.

For example, Figure 2 shows how shared variables are speculatively accessed in the inter-processor communication example that Figure 1 showed. When the processor P_1 arrives to the first barrier at $B_1^a(1)$, which assures the satisfaction of the true-dependency constraint of the accesses to X_2, its execution mode turns to speculative (bright shadow region) to continue the execution. This makes X_2 read before the departure of the barrier at $B_1^d(1)$, but the value obtained by the read is correct because the write/read order satisfies the dependency constraint incidentally and fortunately. The same holds in the read of X_1 by P_2 between $B_2^a(1)$ and $B_2^d(1)$. Thus the processors P_1 and P_2 will not spend idle time to confirm the barrier synchronization.

Similarly, the writes to X_1 and X_2 by P_1 and P_2 are performed after their arrival to second barrier at $B_1^a(2)$ and $B_2^a(2)$, but before the departure at $B_1^d(2)$ and $B_2^d(2)$ assuming the anti-dependency constraints have already been satisfied. Since the assumption is correct again, the speculation successes and the idle time of the second barrier is also eliminated.

As shown in this example, the speculative access aims to hide the latency of the confirmation of a synchronization. Thus if the barrier arrival times vary among processors and, especially, among barriers because of load imbalance and fluctuation, it is expected that the speculative access effectively removes or reduces the idle time. Even in the case of well-balanced load, the latency hiding will be effective if the latency is significantly large because of, for example, a large number of processors involved.

2.2 Speculation Failure by Premature Access

All the accesses shown in the previous section are performed satisfying the dependency constraints fortunately, and thus barrier operations are performed without overheads preserving the semantics of the program.

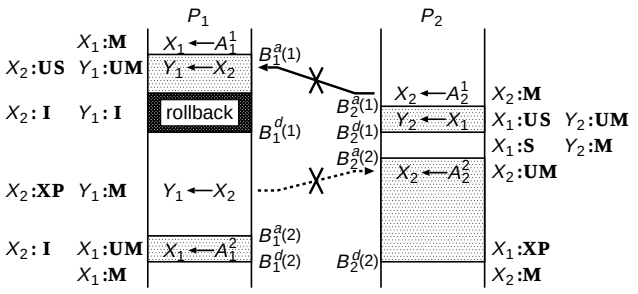

Figure 3. Speculation Failure and Rollback

However, since a speculation is always possible to fail unsatisfying the dependency constraints, we have to take care of the failure in order to preserve the program semantics.

For example, the speculative read of X_2 by P_1 in Figure 3 mistakingly precedes the write to X_2 by P_2 to result an incorrect value obtained. This incorrect value is written into Y_1 and may be propagated to other variables further by the operations by which Y_1 is referred. In this case, at first we have to know somehow that the read of X_2 has performed prematurely, and then have to *rollback* to nullify and to redo all the incorrect computation caused by the premature read of X_2.

These operations might be implemented with a mechanism similar to load/store buffer for dynamic scheduling microprocessors[4, 6, 19]. That is, the addresses of potentially premature loads are kept in an associative memory to check them against the write notifications from other processors, while address/data pairs[1] of stores are held in another associative memory to preserve the computational state at the beginning of the speculation. This mechanism, however, requires not only expensive associative memories that should limit the number of speculative access less than desired to hide long latency of synchronization, but also non-constant burst memory accesses when the speculation is known to be successful[2].

Thus, as described in Section 3 in detail, we devised a mechanism for the failure detection and rollback using a writeback-type coherent cache with a small extension. For the detection, we mark all the cache lines accessed in speculative execution mode, in the period from arrival to departure of a barrier in our example, to indicate that their accesses are potentially unsafe. In the example shown in Figure 3, the state for the cache line containing X_2 becomes **US** (Unsafe Shared) by the speculative read providing it was **S** (Shared) before that. The state of the line for Y_1 also turns to **UM** (Unsafe Modified) corresponding to **M** (Modified). A write notification from another processor to one of these lines with **U** marks means that the accesses to the lines were incorrect possibly. Thus, in our example, the cache of P_1 detects that X_2 was read prematurely when it is notified the write on X_2 by P_2.

Then we rollback the computation exploiting writeback mechanism of the cache as follows. When a line turns to **UM** or **US** from **M**, its old value is written back to memory to preserve the computational state at the beginning of the speculation. Then, on the detection of speculation failure, lines of **UM**[3] are invalidated so that successive accesses will refer the correct values in memory. The computational states other than those in memory, such as in registers, may be saved and restored by the mechanism of, for example, shadow register[18, 21]. After the rollback, the processor will wait until the synchronization completion lest processors mutually and speculatively let others rollback to result a deadlock.

This mechanism with cache has the advantage that a large number of speculative accesses are allowed. Another merit is that a simple functional memory with masked reset makes it possible to invalidate multiple lines in a constant time at the speculation failure, as described in Section 3. This function is also capable of removing all the **U** marks at once when the speculation is known to be successful by the confirmation of the synchronization completion, as done for X_1 and Y_2 in P_2's cache at $B_2^d(1)$.

2.3 Speculative Write to Shared Variable

In the example of Figure 3, P_1 reads X_2 again after its rollback. On the other hand, P_2 successfully passes through the speculative region of the first barrier and performs a speculative write to X_2 before the second barrier synchronization completes. This write, however, breaks the anti-dependency constraint of X_2 because it precedes the read of X_2 by P_1 as shown in the figure.

We could detect the premature write by the read request from P_1 and let P_2 rollback by it as we do in the previous section if we employ a write-invalidate type coherence protocol. However, there is a more efficient way in which the value saved in memory is replied for the read request to the line of **UM**. In this example, P_1 will receive the value A_2^1 saved in memory instead of A_2^2 in the P_2's cache.

This works well so far as P_1 and other processors read X_2 before the departure of the second barrier,

[1] Address and old data pair, alternatively.
[2] At failure, alternatively.
[3] As described later, all the lines in state with **U** marks are invalidated in the implementation model so far.

but will not after that because A_2^1 will become too old. That is, if we have the third barrier not shown in the figure, the value of X_2 should be A_2^2 after the barrier. However, the write notification of the line of X_2 has already been issued and thus P_1 will never have the chance to invalidate (or update) the value of X_2, A_2^1, in its cache.

Thus we give a special state **XP** (eXPiring) to the line that is obtained from memory because another processor's cache has the line of **UM**, and invalidate all the lines of **XP** on the next barrier arrival. In our example, the read of X_2 by P_1 after the arrival $B_1^a(2)$ will miss its cache so that the correct value A_2^2 will be obtained from P_2's cache. Note that the functional memory will perform this multiple invalidation of lines of **XP** in a constant time as well as multiple state transition on speculation success and failure discussed before.

Also note that another processor, say P_3, may have X_2 in its cache when P_2 writes it speculatively[4]. If so, the line in P_3's cache turns to **XP** by the write notification from P_2, instead of being invalidated to avoid unnecessary miss if write-invalidate is in use, or being updated to preserve the correctness in the case of write-update.

3 Implementation Model

3.1 Overview

As discussed in the previous section, specMEM will be implemented by means of a small extension of writeback-type coherent cache. In this section, we show the detail of the implementation model based on a simple coherence mechanism with states **M**, **S** and **I** (Invalid) and write-invalidate protocol. This assumption, however, is just for the sake of conciseness of our discussion, and is not to restrict the base model having additional states and/or employing write-update protocol.

On top of the base model, we introduce three additional cache line states, **UM**, **US** and **XP** corresponding to **M**, **S** and **I** respectively. The transitions between these states are summarized as follows (Figure 4).

1. When a line in ordinary states $\{\mathbf{M}, \mathbf{S}, \mathbf{I}\}$ is read in speculative mode ($r(s)$), the line turns to **US** to indicate speculative read. If the old state is **M**, in addition, the contents of the line is written back to memory to preserve the computational state.

   Similarly, a line turns to **UM** by a speculative write ($w(s)$), and its contents is written back if it was in **M**. This speculative write will be notified to other caches ($W(s)$) to let corresponding lines turn to **XP** to indicate that their contents will be invalid afterward. This transition to **XP** is also taken when a cache tries to obtain a line owned by another cache with state **UM** ($r(n,s)$). In this case, the contents of the line is replied to the requester cache from memory having non-speculative value.

2. When a synchronization is completed (σ^e), all the line in **US** turn to **S**, and those in **UM** to **M**[5]. This transition erases all the **U** marks and thus caches act as ordinary MSI ones until the next speculation.

3. When one of the following occurs, a processor rollbacks to the beginning of the speculation (RB); receipt of a write notification for a line in **US** or **UM**; replacement of a line in **US** or **UM**; or speculative access to a line in **XP**. This makes all the lines in **US** or **UM** turned to **I** so that accesses to them miss the cache to obtain their valid values from the memory[5].

   Note that it is possible to turn lines in **US** to **S** instead of **I** and this modification will improve performance as discussed in Section 4.4. However, this multiple transition requires additional hardware cost for the functional memory (discussed in Section 3.3) in general cases.

4. On the next synchronization (σ^b), lines in **XP** may have expired values. Thus they turn to **I** in order to obtain correct values.

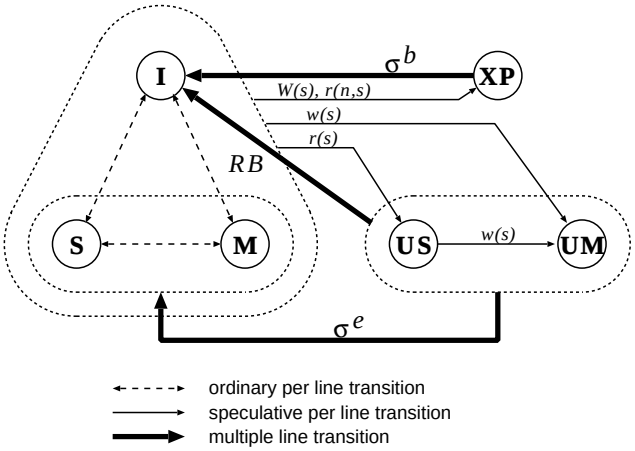

Figure 4. Overview of State Transition

[4] P_1 does not, because the corresponding line has been invalidated at the rollback.

[5] As shown in Table 1 later, all the lines in **XP** turn to **I** but this transition is not essentially required.

Table 1. State Transition of Cache Line

from	to I	to S	to M	to US	to UM	to XP
I	$r(s,s)$+RB, σ^b, σ^e, RB	$r(n,n)$	$w(n)$+W	$r(s,n)$	$w(s)$+W	$r(n,s)$
S	$W(n)$, v	$r(n)$, σ^b, σ^e, RB	$w(n)$+W	$r(s)$	$w(s)$+W	$W(s)$
M	$W(n,c)$, v+WB	$R(c)$	$r(n)$, $w(n)$, σ^b, σ^e, RB	$r(s)$+WB	$w(s)$+WB	$W(s,c)$
US	$W(*)$+RB, v+RB, RB	σ^e	—	$r(s)$	$w(s)$+W	—
UM	$W(*,m)$+RB, v+RB, RB	—	σ^e	—	$r(s), w(s), R(m)$	—
XP	$r(s)$+RB, $w(s)$+RB, $W(n)$, RB, σ^b, σ^e, v	—	$w(n)$+W	—	—	$r(n)$, $W(s)$

Table 2. Encoding of Cache Line State

state	$b_2b_1b_0$	σ^b	σ^e	RB
I	000	**I** (000)	**I** (000)	**I** (000)
S	001	**S** (001)	**S** (001)	**S** (001)
M	010	**M** (010)	**M** (010)	**M** (010)
XP	100	**I** (000)	**I** (000)	**I** (000)
US	101	—	**S** (001)	**I** (000)
UM	110	—	**M** (010)	**I** (000)

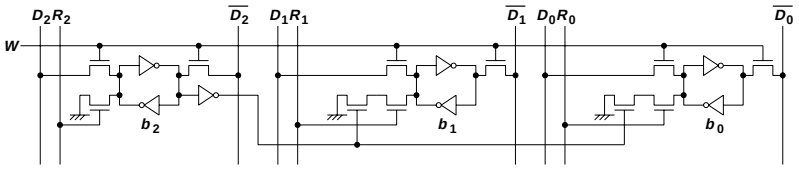

Figure 5: Memory Cells for State Bits of a Line

3.2 State Transition in Detail

The complete definition of the state transition is given in Table 1, in which each of the following symbol represents an event triggering a transition or, if the symbol is prefixed by a +, an action taken at a transition.

- $r(\{s|n\}\,[,\{s|n\}])$ is a read by the processor owning the cache. The first argument indicates whether the processor is in speculative mode (s) or not (n). The optional second argument, for the case of miss, indicates whether there is a **UM** cache line in the system (s) or not (n).
- $w(\{s|n\})$ is a write by the owner processor. The argument indicates whether the processor is in speculative mode (s) or not (n).
- $R(\{c|m\})$ is a read request from a processor other than the owner which is responsible for the reply to the request. The argument indicates whether the line is provided from the cache (c) or memory (m) as the reply.
- $W(\{s|n|*\}\,[,\{c|m\}])$ is a write notification from a processor other than the owner. The first argument indicates whether the writer processor is in speculative mode (s) or not(n), or mode independent ($*$). If the cache is responsible for providing the line, the optional second argument indicates whether the line is from the cache (c) or memory (m). $+W$ means a write notification from the owner possibly issued at a transition triggered by $w(\{s|n\})$.
- v is a replacement of the line by another.
- σ^b is the beginning of a synchronizing operation, such as $B_i^a(j)$.
- σ^e is the end of a synchronizing operation, such as $B_i^d(j)$.
- RB is a rollback. $+RB$ means that the state transition is accompanied by a rollback.
- $+WB$ means that the state transition is accompanied by a writeback.

Note that a rollback may be triggered by an event not shown in the table. For example, a memory access exception probably caused by a pointer variable accessed prematually should be included in the events, as well as a TLB miss for the performance sake.

3.3 Implementing with Functional Memory

The state transitions triggered by σ^b, σ^e and RB are made for multiple lines. This simultaneous state transition of multiple lines is performed by a functional memory that has the following simple functions.

1. $reset(b_r)$ to turn the bit b_r in all the words into 0.
2. $masked\_reset(b_m,b_r)$ to turn the bit b_r in the words, whose bit b_m is 1, into 0.

With these functions and the encoding of cache line states shown in Table 2, the multiple line transition for each trigger event is implemented as follows.

σ^b : $reset(b_2)$;

σ^e : $reset(b_2)$;

RB : $masked\_reset(b_2, b_1)$; $masked\_reset(b_2, b_0)$;
$reset(b_2)$;

Figure 5 shows an example of the memory cell configuration for the functional memory. The ordinary access to the bit b_i is controlled by the word-line W and bit-line D_i, while (masked) reset is performed by charging the line R_i. Since memory cells for the three state bits in a cache line tag will only have additional seven transistors to CMOS SRAM, almost equivalent to one bit addition, the hardware implementation cost should be acceptably small. Power consumption on reset will be also acceptable if we make the reset time significantly longer than the ordinary access time. Since the reset time only affects the cost of operations σ^b, σ^e and RB, the system performance should not be sensitive to it.

4 Experiment

4.1 Hardware Model for Evaluation

For a preliminary performance evaluation of specMEM, we constructed a simulator of a SPARC-base centralized shared memory multiprocessor system having architectural parameters shown in Table 3. Although the coherent cache of the system is MESI-base (i.e. including Exclusive state like Illinois protocol[22]) rather than MSI-base described in the previous section, there are neither complication to define state transitions to/from **E** and **UE** nor additional transistors to implement them.

In order to simplify the simulator and the analysis of the results, we adopt a simple processor model with single instruction issue, in-order execution, and sequential consistency memory model[12]. As for synchronization, we assume the system has a simple hardware barrier mechanism like SBM[16]. An additional instruction notifies the barrier arrival to the hardware mechanism and saves computational states in registers. On the last arrival to the barrier, each processor and cache is notified it with the delay (the number of cycle) shown in the table.

Note that it is not essential that we adopted the configuration and the architectural parameters for a simple and small scale SMP. Instead, it is just for obtaining performance data promptly and the application of specMEM is not restricted to such a system. For example, a relaxed memory model such as weak consistency[1] is not only easily applicable but also will be implemented efficiently in specMEM. That is, while ordinary implementations have to confirm the completion of preceding memory writes on the barrier arrival, specMEM may delay it to the departure to hide write latency[6]. This possibly causes that a processor reads a location after a barrier arrival prior to the completion of a pre-barrier write performed by another processor. However, this write/read order reversal will be detected and causes rollback because all the post-barrier accesses are speculative.

Table 3. Architectural Parameters for Evaluation

# of processors	4	cost/penalty (# of cycles)	
processor ISA	SPARC V8	ordinary instructions	1
cache		barrier synchronization	+10
capacity	64 KB	state saving	+5
line size	16 B	rollback	+10
associativity	4 way	cache miss	
coherence	MESI	memory→cache	+20
		cache→cache	+10
		invalidation	+5
		writeback	+10

Our specMEM is also easily applicable to a large scale system such as a directory based distributed shared memory system. In fact, the essential modification to a distributed cache coherence protocol is only in the existence of **UM** owner that cannot reply to the read-requester directly but has to notify the directory to give the content of the memory to the requester.

4.2 Workloads

The following three workloads are chosen from SPLASH-2[23] benchmarks[7] to measure their execution time (the number of cycle) with and without speculative accesses.

LU Decomposition Since computational loads are not balanced well and the heaviest load shifts from a processor to another, it is expected that the speculation will be effective.

FFT Loads are almost completely balanced, and thus there will be almost no chance of the speculation. Therefore, this will examine if our mechanism is *neutral*.

Radix Sort Loads are not balanced well but a processor always has the heaviest one. Thus, speculative accesses by other processors having lighter loads will not contribute to performance improvement, but could show a bad influence of our mechanism if any. This is a tougher test of the neutrality.

[6] The completion of preceding reads has to confirmed on the arrival.

[7] The sizes of problems are reduced because of the space limit of our simulator.

4.3 Performance Results

Figure 6 shows the normalized execution times, letting those of non-speculative ones be 100, of three programs. It also shows the breakdown of the execution in each processor and the frequency of rollback with respect to barriers. The rollback cost includes the re-execution after rollback, and the ache miss cost includes the penalty for shared write and writeback for the computational state saving.

4.3.1 LU Decomposition

As expected, because of the significant reduction of idle times owing to the speculative accesses, performance improvement of 13 % is achieved. In this program, $P3$ has the heaviest load in total because it is responsible for the decomposition of the matrix block at the right-bottom corner, but each of other processors is also the last arrival of a barrier when it is the decomposer of another diagonal block. Thus, as discussed in Section 2.1 and shown in Figure 2, load distribution fluctuates to make the speculation mechanism work efficiently.

Readers may notice that the idle time is not completely removed even in $P3$ with the heaviest load. This is because a barrier arrival has to follow the previous barrier departure and thus speculative regions cannot be overlapped. The removal of this restriction, which will significantly increase the hardware cost to maintain multiple speculative states but will improve the performance by about 7 %, is an issue of our further study.

It is also noticeable that the rollback costs including that for re-execution are not small in $P1$ and $P2$, but these costs do not directly affect the total performance in this program. The cache miss penalties, on the other hand, are doubled by the speculation in all the processors including $P3$ that has the heaviest load. This increment of the miss penalty is about 4% of the total execution time. We will discuss about this problem later.

4.3.2 FFT

As also predicted, the performance is not improved by the speculation at all because the loads are almost completely balanced. Although specMEM would hide the inherent latency of barrier synchronization, both our simulation parameter of the latency (10 cycle) and the number of barriers (12) are too small to make the effect of the hiding significant. However, if the inherent latency is enlarged because of larger number of processor and/or lack of hardware support[8], a significant effect should be observed for a program having a cer-

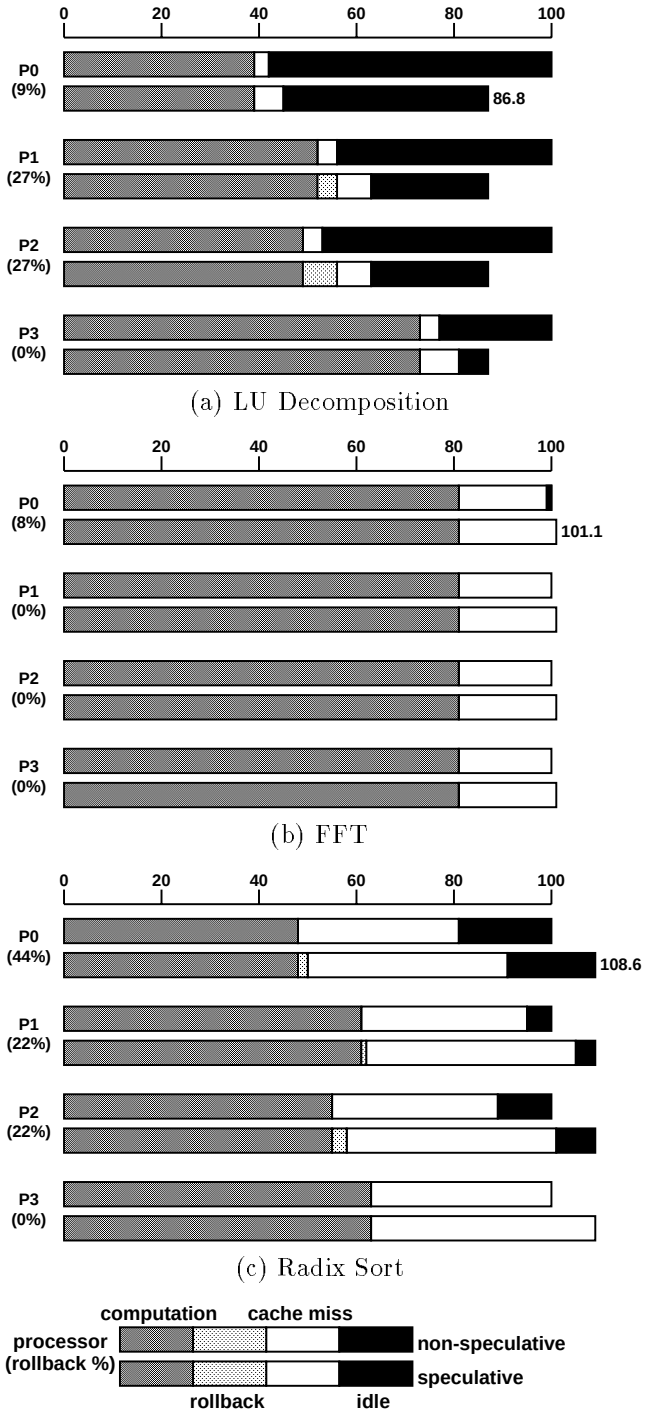

Figure 6. Performance Results

[8]The departure of a barrier should be visible by caches but the hardware cost for this requirement is much smaller than a complete hardware support of the barrier mechanism itself.

tain number of barriers.

The purpose of the evaluation, on the other hand, is accomplished by showing specMEM is almost neutral to such a well balanced parallel computation. A slight performance degradation, caused by a small number of cache miss increment, will be discussed later.

4.3.3 Radix Sort

Because of the tree structure of the program, $P3$ is always the last arrival of critical barriers. Thus it was predicted that the speculation would be ineffective. The annoying behavior is the increment of cache miss penalty, especially in the critical path executed by $P3$, which degrade the performance a little bit too largely to say our mechanism is at least neutral.

As discussed later, the reason of the penalty increment in $P3$ is 25 % growth of cache misses caused by a type of false sharing.

4.4 Analysis of Cache Miss Penalty

4.4.1 Penalty Caused by Rollback and Writeback

Speculative execution may change the timing and number of memory accesses to result the enlargement of cache miss penalty. One reason of the penalty growth is that of the number of cache misses, and the other is that of memory and bus traffics added by the writebacks for state saving.

In LU decomposition, we found that the number of shared bus accesses of specMEM is about one and a half of the system without speculation. Especially, the number is almost doubled in $P2$ that pays the highest rollback cost. This is mainly due to the large miss rate 2.1 % in re-execution phases, which is much larger than 0.6 % in other phases because all the speculatively accessed lines are invalidated by rollback. The other reason making bus busier is writeback for state saving, which increases the number of bus accesses by 15 %.

These two reasons affect the execution time not only of the processor that performs rollback or writeback, but also of other processors. In fact, although the miss rate of $P3$ with the heaviest load is almost unchanged by the speculation, its miss penalty is doubled as in $P2$ and others to result 4 % performance degradation. Thus we are now studying the following improvements to reduce bus and memory traffics[15].

- It is unnecessary to invalidate lines of state other than **UM** on rollback. For example, if we replace the multiple line transition on rollback shown in Section 3.3 with

$$RB : masked\_reset(b_2, b_1);\ reset(b_2);$$

lines in **US** will turn to **S** instead of **I**. Although this solution requires an additional bit or transistors for the mask operation in general cases with more base states, the hardware cost may be justified if this reason is dominant in some other applications.

- It is also unnecessary to writeback a line on the transition from **M** to a state other than **UM**. If we add a speculative state, namely **UD**, indicating that the line is dirty and speculatively read, the frequency of writeback will be reduced because only the transitions $\{\mathbf{M}, \mathbf{UD}\} \rightarrow \mathbf{UM}$ requires it.

- Attaching a non-speculative secondary cache will drastically reduce the bus traffics in re-execution phase and for writeback. A line in the secondary cache may have only one state additional to base states, namely **U**, indicating that the corresponding line in the speculative primary cache is possibly in **UM**. If the line in the primary is really in **UM**, its counterpart in the secondary has the value written back from the primary. Since the real state of the line in **U** is defined by its primary counterpart, multiple line transition is not necessary for the secondary cache.

4.4.2 Penalty Caused by False Sharing

False sharing problem, due to which our mechanism fails to be neutral for Radix Sort, is more difficult to solve. In Radix Sort, we observed the following miserable scenario that increases the number of misses in $P3$ by 25 %.

1. The cache of $P3$ has a line L for array element $x[i]$ and $x[i+1]$ in state **M**.
2. Before $P3$ arrives a barrier, another processor, say $P2$, speculatively writes $x[i]$ missing its cache to turn the state L in $P3$'s cache from **M** to **XP**.
3. $P2$ is notified a write from $P3$ resulting rollback and invalidates L.
4. $P3$ arrives the barrier to invalidate L in **XP**.
5. $P3$ writes $x[i+1]$ on L resulting a write miss.
6. $P2$ writes $x[i]$ on L in its re-execution phase missing its cache too.

Note that there are three misses for L including one by $P3$. In non-speculative execution, on the other hand, there is only one miss by $P2$. Worse enough, since $P3$ generates the source of i in its work prior to the barrier, it is even observed that $P2$ does not access $x[i]$ in non-speculative execution.

These observations reveal the limit of the blind speculation lead only by hardware. At least we have to have a compiler that switches the speculation off if it obviously (or likely) brings such a disaster. In terms of hardware mechanism, an adaptive switching of the speculation based on the success/failure history may be useful to prevent the system from too aggressive.

5 Related Works

5.1 Fine-Grain Communication

As stated in the introductory section, specMEM aims to reduce the synchronization overhead in the *coarse-grain* communication through shared memory. Another approach to reduce the communication cost, namely *fine-grain* approach, is to minimize memory locations associated to a synchronizing operation so that the access inhibition is applied only to those required essentially.

I-structure[2] and its shared memory implementation[8], in which the synchronization tag attached to each memory word can be examined by the load operation to obtain the data part, is a good example of this approach. This mechanism will completely hide the latency of a synchronization if the temporal distance between write and read of a shared variable is enough large, as specMEM expects. Similar communication can be performed with release consistency (RC)[5] or entry consistency (EC)[3], if we associate a synchronization variable to a small chunk of shared variables and hardware allows enough number of incomplete accesses.

Various researches on barrier synchronization mechanisms also aim at the efficient fine-grain communication. The *fuzzy barrier*[9] has certain similarity to specMEM because it allows *safe* operations executed between the barrier arrival and departure. However, specMEM has an advantage of the fuzzy barrier because it allows *any* operations in the region. Splitting arrival and departure in *elastic barrier*[13] is more aggressive because it allows that a region between them is overlapped to other barrier regions. Thus it will hide synchronization latency efficiently as RC and EC does by making each barrier region small.

These mechanisms, however, require not only dedicated hardware for latency hiding but also large efforts of programmers and/or compilers. Assume the following simple loop:

```
parallel for (p=0;p<2;p++) {
    for (i=0;i<n;i++) a[p][i]=... ;
    for (i=0;i<n;i++) ...=a[1-p][i] ; }
```

In fine-grain approaches, two `for` loops must be hashed finely and appropriately to hide the latency of the synchronization to assure the write and read order of the elements of array `a`. In our coarse-grain speculative approach, the latency will be hidden by simply putting a barrier between two loops.

5.2 Coherent Cache and Speculation

Researches on coherent cache and speculation are divided into two groups; one is for adaptive coherence control, and the other is for speculative accesses that specMEM aims at. The former aims to predict the most suitable coherence control operation based on the history per instruction[10] or per memory block[11, 14]. Although the predicted operation is performed *speculatively* before it is known to be really suitable, the operation is always safe with respect to the program semantics. Thus the aim and mechanism is different from those of specMEM, but the prediction methods could be applied to the adaptive speculation mentioned in Section 4.4.

An example of later group is Gniady's SC++[6] that aims to reorder memory accesses dynamically and speculatively even when a program requires sequential consistency (SC)[12]. As specMEM does, SC++ assumes the completion of synchronization hidden in SC program and detects speculation failure by a write notification. However, it uses associative buffers to trace speculative accesses and thus hardly hide long latency of synchronization.

The other example is Gopal's *speculative versioning cache* (SVC)[7] for hierarchical multithreaded execution of a sequential program, which was proposed independently of our first proposal[17]. SVC exploits coherent cache for the failure detection and computational state preservation instead of associative buffers used in other related researches[4].

The obvious difference between SVC and specMEM is that SVC is for sequential programs while specMEM is for parallel ones, and it leads an essential difference in the implementation difficulty. That is, since the hierarchical execution creates small parallel threads executed speculatively, SVC has to maintain multiple versions of the value in a memory location, one of which should be chosen according to the program order of a thread accessing the location. Thus a centralized logic for the version control is required limiting the system to a bus-connected SMP with a few processor although it is sufficient for the aim of SVC. Opposite to it, the coarse synchronization allows specMEM to maintain only two versions. Therefore, specMEM is easily applicable to a directory based large scale distributed shared memory system as discussed in Section 4.1.

6 Conclusions

We proposed an efficient mechanism specMEM to execute memory accesses following a synchronizing operation speculatively. The implementation of specMEM only requires a small extension of coherent cache with a simple functional memory to perform the speculation failure detection, computational state preservation, and other operations on the speculation success and failure in a constant time. The effectiveness of specMEM for the programs with load fluctuation is proved by the experiment showing 13 % performance improvement of LU decomposition. The techniques for further performance improvement are discussed based on the analysis of the cache miss penalty increased by the speculation.

The experiment also shows the necessity of the cooperation from software side to inhibit the speculation if it is too dangerous as in Radix Sort. This cooperation should be mandatory when we apply specMEM to a program with frequent lock operations.

Our urgent future works are for the reduction the cache miss rate by the discussed techniques and for the compiler supported speculation switching. We also plan to evaluate specMEM applying it to a larger scale distributed shared memory system with more varieties of programs including those with locks rather than barriers.

References

[1] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *Proc. 17th Intl. Symp. on Computer Architecture*, May 1990.

[2] Arvind, R. S. Nikhil, and K. Pingali. I-structure: Data structures for parallel computing. *ACM Tans. on Prog. Lang. and Syst.*, 11(4):598–632, Oct. 1989.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. IEEE COMPCON*, 1993.

[4] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering memory references. In *Proc. Intl. Symp. on High-Performance Computer Architecutre*, Mar. 1998.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 15–26, May 1990.

[6] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proc. 26th Intl. Symp. on Computer Architecture*, May 1999.

[7] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proc. Intl. Symp. on High-Performance Computer Architecutre*, Mar. 1998.

[8] M. Goshima, S. Mori, H. Nakashima, and S. Tomita. The intelligent cache controller of a massively parallel processor JUMP-1. In A. Veidenbaum and K. Joe, editors, *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 116–124. IEEE, Oct. 1997.

[9] R. Gupta. The fuzzy barrier. In *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, Apr. 1989.

[10] S. Kaxiras and J. R. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *Proc. 5th Intl. Symp. on High-Performance Computer Architecture*, Jan. 1999.

[11] A. C. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proc. 26th Intl. Symp. on Computer Architecture*, May 1999.

[12] L. Lamport. How to make a multiprocessor computer that correctly execute multiprocessor programs. *IEEE Trans. on Computers*, 28(9):690–691, Sept. 1979.

[13] T. Matsumoto, T. Tanaka, T. Moriyama, and S. Uzuhara. MISC: A mechanism for integrated synchronizatin and communication. In *Proc. Intl. Conf. on Parallel Processing*, volume 1, pages 161–170, Aug. 1991.

[14] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Proc. 25th Intl. Symp. on Computer Architecture*, May 1998.

[15] H. Nakashima, T. Sato, H. Matsuo, and K. Ohno. Implementation issues of the speculative memory access mechansim specMEM. `http://www.para.tutics.tut.ac.jp/~nakasima/papers/specmem-imp.ps.gz`, Jan. 2000.

[16] M. T. O'Keefe and H. G. Dietz. Hardware barrier syhchronization: Static barrier MIMD (SBM). In *Proc. Intl. Conf. on Parallel Processing*, volume 1, pages 35–42, Aug. 1990.

[17] T. Satoh and H. Nakashima. Proposal of a speculative memory access correlative to synchronizing operations. In *IPSJ SIG Notes, 98-ARC-129*, pages 19–24, May 1998. (In Japanese. English version is available at `http://www.para.tutics.tut.ac.jp/~nakasima/papers/specmem-98.ps.gz`).

[18] K. Seo and T. Yokota. Pegasus: A RISC processor for high-performance execution of Prolog programs. In *Proc. of Intl. Conf. on Very Large Scale Integration*, pages 261–274, 1987.

[19] J. E. Smith. Dynamic instruction scheduling and the astronautics ZS-1. *Computer*, 22(7):21–35, July 1989.

[20] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *Proc. ASPLOS'89*, pages 290–302, Apr. 1989.

[21] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 344–355, May 1990.

[22] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):14–24, June 1990.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methdological considerations. In *Proc. ISCA'95*, pages 24–36, June 1995.