Improved Implementations of the Speculative Memory Access Mechanism specMEM

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

Abstract

In order to reduce the overhead of synchronizing operations of shared memory multiprocessors, we proposed 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.

Although we reported in [4] that specMEM achieves significant execution time reduction, for example 13% for LU decomposition, we also observed that it may be implemented more efficiently. This paper discusses about more efficient implementations of specMEM with an extra cache state and/or a non-speculative secondary cache.

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. 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 $B_1^I(1)$ because it follows the arrival of $P_1$ at $B_2^I(1)$. Thus the time from the arrival $B_1^I(1)$ to the departure $B_2^I(1)$ (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 $B_2^I(1)$, the true-dependency constraint of the read of $X_2$ by $P_1$ is satisfied at a certain point prior to the departure time $B_2^I(1)$. 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.

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 proposed a speculative memory access mechanism named specMEM[4], in which an access following a synchronizing operation is performed prior to the confirmation of synchronization assuming the satisfaction of data dependency constraint.

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.

The performance data obtained by simulation running benchmark programs in SPLASH-2 showed that the execution time of LU decomposition, in which the length of period between a pair of barriers signif-
stantly varies because of the fluctuation of computational load, is reduced by 13%. The evaluation result, however, also revealed a performance bottleneck due to the heavy memory traffic. As discussed in this paper, this problem will be solved by adding a speculative cache state and/or introducing a non-speculative secondary cache.

The rest of this paper consists of the followings: Section 2 outlines how the specMEM works in the case of successful speculation and of failure. Section 3 gives its basic implementation model with a small extension of coherent cache. Section 4 discusses about the improved implementation with an extra cache state and non-speculative secondary cache. Finally we conclude the paper summarizing future works in Section 5.

2 Overview of specMEM

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 $B_1(1)$, 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 $B_1(1)$, 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 $B_2(1)$ and $B_2(2)$. 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 $B_1(2)$ and $B_2(2)$, but before the departure at $B_1(2)$ and $B_2(2)$ 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. 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 mistakenly 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 vari-
ables 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\[1, 2, 6\]. 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 write-back-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 $P_2$.

When $P_1$ detects that $X_2$ was read prematurely, it rolls back its state to the previous correct state. This is done by invalidating all the cache lines containing $X_2$ and $Y_2$. The invalidated cache lines are then forced to be re-read from memory, which is done by the processor that caused the invalidation.

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 $A_{12}^1$ saved in memory instead of $A_{12}^0$ 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, but will not after that because $A_{12}^1$ will become too old. That is, if we have the third barrier not shown in the figure, the value of $X_2$ should be $A_{12}^0$ 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 $X_2$, $A_{12}^1$, in its cache.

\[1\]Address and old data pair, alternatively.

\[2\]At failure, alternatively.
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 \( B^2_1(2) \) 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 \( P_2 \)'s cache turns to XP by the write notification from \( P_3 \), 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 Basic 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\(^3\).

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 \( \{M, S, I\} \) 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 (\( \sigma^e \)), 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 rolls its execution back 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\(^6\).

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 4.2. 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 (\( \sigma^s \)), lines in XP may have expired values. Thus they turn to I in order to obtain correct values.

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.

\(^4\)\( P_1 \) 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

<table>
<thead>
<tr>
<th>from</th>
<th>I</th>
<th>S</th>
<th>M</th>
<th>US</th>
<th>UM</th>
<th>XP</th>
</tr>
</thead>
<tbody>
<tr>
<td>r{s[n]}{s[n]}</td>
<td>r(s, n)</td>
<td>r(s, n)</td>
<td>w(n) + W</td>
<td>r(s, n)</td>
<td>w(s) + W</td>
<td>r(m, s)</td>
</tr>
<tr>
<td>W(n), v</td>
<td>r(n), v</td>
<td>r(n), v</td>
<td>r(n), v</td>
<td>r(n), v</td>
<td>r(n), v</td>
<td>r(n), v</td>
</tr>
<tr>
<td>W(n), v + WB</td>
<td>R(c)</td>
<td>r(n), w(n), s^, s^, RB</td>
<td>r(n), w(n), s^, s^, RB</td>
<td>r(n), w(n), s^, s^, RB</td>
<td>r(n), s^, s^, v</td>
<td>W(n)</td>
</tr>
<tr>
<td>W(n), v + WB</td>
<td>R(c)</td>
<td>r(n), w(n), s^, s^, RB</td>
<td>r(n), w(n), s^, s^, RB</td>
<td>r(n), w(n), s^, s^, RB</td>
<td>r(n), s^, s^, v</td>
<td>W(n)</td>
</tr>
<tr>
<td>W(s, m) + RB, v + RB, RB</td>
<td>s^</td>
<td>s^</td>
<td>s^</td>
<td>s^</td>
<td>s^</td>
<td>s^</td>
</tr>
<tr>
<td>w(s[n])</td>
<td>w(n) + W</td>
<td>w(n) + W</td>
<td>w(n) + W</td>
<td>w(n) + W</td>
<td>w(n) + W</td>
<td>w(n) + W</td>
</tr>
<tr>
<td>W(n), RB, s^, s^, v</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( r\{s[n]\}\{s[n]\} \) 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 an 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 (s). 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.

- \( s^\hat{b} \) is the beginning of a synchronizing operation, such as \( B^\hat{b}_f(j) \). In general, it corresponds to the start of an acquire operation to examine if the following operations may be executed with respect to the synchronization point. To execute the operations following \( s^\hat{b} \) speculatively, it is desirable that the examination is performed by a simple hardware mechanism working concurrently with the processor. If such a mechanism is attached, \( s^\hat{b} \) is an instruction (sequence) to activate it and to notify the cache of the start of speculation. Otherwise, \( s^\hat{b} \) just notifies the cache of the start of speculation.

- \( s^\hat{c} \) is the end of a synchronizing operation, such as \( B^\hat{c}_f(j) \). In general, it corresponds to the end of acquire operation at which the synchronization is confirmed. If the hardware mechanism for the examination of synchronization is attached, \( s^\hat{c} \) may not have a corresponding instruction in the processor code but is an event signal transmitted from the mechanism to the cache. Otherwise, it must be an instruction (sequence) to notify the cache of the end of speculation preceded by that for the examination performed by the processor itself.

- \( RR \) is a rollback. \( +RR \) 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 prematurely 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 \( s^\hat{b} \), \( s^\hat{c} \) and \( RR \) 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. \( \text{reset}(b_r) \) to turn the bit \( b_r \) in all the words into 0.
2. \( \text{masked\_reset}(b_m, b_r) \) 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.

\[
\begin{align*}
\sigma^\hat{b} & : \text{reset}(b_2); \\
\sigma^\hat{c} & : \text{reset}(b_2); \\
RR & : \text{masked\_reset}(b_2, b_1); \text{masked\_reset}(b_2, b_0); \text{reset}(b_2);
\end{align*}
\]

Figure 5 shows an example of the memory cell configuration for the functional memory. The ordinary access
to the bit $b_1$ is controlled by the word-line $W$ and bit-line $D_2$, while (masked) reset is performed by charging the line $R_2$. 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 $\sigma^+, \sigma^-$ and $RB$, the system performance should not be sensitive to it.

4 Improved Implementation

4.1 Drawbacks of the Basic Implementation

In the basic design, we added speculative states $UM$, $US$ and $XP$ corresponding to the base states $M$, $S$ and $I$. In general, on top of a coherent cache having base states $\{s_0=I, s_1, \ldots, s_n\}$, $\text{specMEM}$ can be implemented by adding $\{Us_0=XP, Us_1, \ldots, Us_n\}$. Since the number of additional speculative states is at most same as that of base states, we need only one additional bit to represent base and speculative states.

This design, however, has the following drawbacks.

- On rollback, lines in state $US$ are unnecessarily invalidated. In general, a line in $US_i$ which has been read speculatively but not written are unnecessarily invalidated because a sequence of (masked) reset cannot distinguish $US_i$ from $UM$ that has to be invalidated.

- Due to the multiple invalidation of $US_i$ lines on rollback, they have to be clean to avoid multiple write-back. Thus dirty lines read speculatively but not written are unnecessarily written back to memory. For example, a speculative read on a line in $M$ turns its state to $US$ with write-back.

The first problem may cause high miss rate in the re-execution phase after rollback, while the second makes memory traffic heavy in the speculative execution phase. For example, the specMEM applied to an SMP of four processors achieves 13% execution time reduction for the LU decomposition in SPLASH-2[8] as shown in Figure 6, but its effectiveness is limited by almost doubled miss penalty.

The breakdown of the bus accesses shown in Figure 7 makes it clear how the drawbacks degrade the performance. The number of bus accesses performed by $P_1$ and $P_2$, which are most frequent speculation failures, is almost doubled. The other significant increment of the bus accesses is caused by the write-back on the transition from a base state to a speculative
state. By these two reasons the bus and memory traffics are made heavy resulting higher penalty of the bus accesses by $P_3$ that executes the critical path and never fails speculation. Thus if we removed the increment of bus accesses completely, specMEM could achieve about 20% speedup for the LU decomposition.

4.2 Alternative Cache Design

The problems shown above will be partly solved if we have one more additional state, namely SM (Speculatively Modified), that has the same role as UM in the basic design. That is, we redefine $Us_i$, other than XP but including UM, meaning that the line is speculatively read and thus may cause rollback if modified but its data value is correct. By this introduction of SM and redefinition, $Us_i$ lines may turn to $s_i$ on rollback and be dirty.

For example, the state transition of the MSI base cache with SM is as shown in Figure 8. The differences from the basic implementation shown in Figure 4 are as follows.

1. When a line in M is read speculatively ($r(s)$), it turns to UM, which now means dirty, exclusive and unsafe but correct, rather than US. Since this state transition is not accompanied by write-back, the speculative read causes neither bus nor memory accesses. The UM line may turn to US when it read by a non-owner processor ($R(e)$).

2. A speculative write ($w(s)$) to a line in any state makes the line turned to SM. If the line was dirty (i.e. in M or UM), it is written back to memory to preserve the computational state.

3. When a synchronization is completed ($\sigma^e$), all the U marks are erased and all the SM lines turn to M.

4. On rollback (RB), all the SM lines are invalidated. However, US and UM lines, except for the target of the write notice triggering the rollback, are simply turn to their non-speculative counterpart and thus are kept valid. Therefore, accesses to these lines in re-execution phase will not miss the cache.

The cost we have to pay for this alternative design is one more additional bit to each cache line tag. In general, if the states of the base cache are encoded in $k$ bits $b_{k-1} \ldots b_0$, we have to add two bits $b_{k+1}$ and $b_k$ to represent speculative states. If the state $s_i$ of the base cache has the code $(c_{k-1} \ldots c_0)$, its code in the specMEM cache is $(0b_{k-1} \ldots b_0)$ and that of its speculative counterpart $Us_i$ is $(0c_{k-1} \ldots c_0)$. The state SM is represented as $(1c_{k}^{\sigma} \ldots c_0^{\sigma})$ where $(c_{k-1}^{\sigma} \ldots c_0^{\sigma})$ is the code for M.

Without loss of generality, we can assume I is encoded as $(0 \ldots 0)$. Thus we can implement multiple state transition by the functional memory as follows.

$RB : \text{masked\_resel}(b_{k+1}, b_{k-1});$

$\ldots; \text{masked\_resel}(b_{k+1}, b_0);$

$\text{resel}(b_{k+1}); \text{resel}(b_k);$

$\sigma^e : \text{resel}(b_{k+1}); \text{resel}(b_k);$

$\sigma^b : \text{resel}(b_k);$

Note that the masked reset operations for $b_k$ may be omitted if $c_k^{\sigma} = 0$.

4.3 Non-Speculative Secondary Cache

The performance of a system with specMEM will be improved by attaching a secondary cache as well as usual shared memory multiprocessors. In fact, the effect of the secondary cache will be larger than usual because it acts as the computational state preserver. A speculative secondary cache, however, should be too expensive because we have to have a (unlikely on-chip) functional memory of a capacity much larger than (likely on-chip) primary cache.

Thus the secondary cache must be non-speculative. That is, each line in the secondary cache must have a correct value and, if it is not included in the primary cache, it must be safe. The second requirement is easily satisfied if:

- the secondary forwards the write notification to the primary for a line (potentially) included in the primary, as done in usual systems; and
• the replacement of \( U_s \) and SM line in the primary causes rollback, as done in one level cache specMEM.

The mechanism to satisfy the first requirement is a little bit complicated because of the behavior of SM (or UM in the basic design) and XP lines. For SM lines, the easy part of the mechanism is to notify secondary that a primary line turns to SM on a speculative write. It is also easy to write a dirty line back to secondary on its transition to SM if the primary is write-back type.

The hard part is due to that the secondary hardly cope with the multiple transition \( SM \rightarrow M \) on \( \sigma^j \) and \( SM \rightarrow I \) on \( RB \) because the secondary is implemented by an ordinary (non-functional) memory. Thus the secondary must have a speculative state named \( U \) representing that the corresponding primary line may be in SM. (This means a non-U line in the secondary cannot have its SM correspondent in the primary.) Since the secondary does not know whether the primary line corresponding to a U line is really in SM, it forwards read/write request from other caches to the primary. The reply to the request is the primary line if it is in \( M \), while the secondary line is replied otherwise.

As for XP lines, it is also hard for the secondary to cope with the multiple transition \( XP \rightarrow I \) on \( \sigma^j \), \( \sigma^x \), and \( RB \). Thus we also need a speculative state named \( X \) representing that the corresponding primary line may be in XP. The meaning of \( X \) is almost same as \( I \) except that a write request to the line is forwarded to the primary.

Figure 9 summarize the state transition of an MSI-type non-speculative secondary cache. As shown in the figure, all the transitions are per-line and per-access allowing us to implement the secondary cache with ordinary non-functional memory.

5 Conclusions

We discussed about two implementation issues to improve the performance of our speculative memory access mechanism specMEM. The first one is to add one more speculative state to avoid unnecessary invalidation and write-back. The other is to introduce a non-speculative secondary cache that acts as the computational state preserver. Both techniques will reduce or remove memory traffic in speculative execution and re-execution phases.

Our urgent future works are to evaluate specMEM again applying these implementation techniques.

![Figure 9. State Transition of Non-Speculative Secondary Cache](image)

References


