Novel Checkpointing Algorithm for Fault Tolerance on a Tightly-Coupled Multiprocessor

Dwight Sunada
David Glasco[1]
Michael Flynn

Technical Report: CSL-TR-99-776

January 1999 (Revised on April 1999)

‘Dr. David Glaxo is affiliated with the Austin Research Laboratory at International Business Machines, Inc.

Novel Checkpointing Algorithm for Fault Tolerance on a Tightly-Coupled Multiprocessor

Dwight Sunada
David Glasco
Michael Flynn

Technical Report: CSL-TR-99-776

January 1999

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
William Gates Building, A-408
Stanford, California 94305-9040
<e-mail: pubs@shasta.stanford.edu>

Abstract

The tightly-coupled multiprocessor (TCMP), where specialized hardware maintains the image of a single shared memory, offers the highest performance in a computer system. In order to deploy a TCMP in the commercial world, the TCMP must be fault tolerant. Researchers have designed various checkpointing algorithms to implement fault tolerance in a TCMP. To date, these algorithms fall into 2 principal classes, where processors can be checkpoint dependent on each other. We introduce a new apparatus and algorithm that represents a 3rd class of checkpointing scheme. Our algorithm is distributed recoverable shared memory with logs (DRSM-L) and is the first of its kind for TCMPs. DRSM-L has the desirable property that a processor can establish a checkpoint or roll back to the last checkpoint in a manner that is independent of any other processor. In this paper, we describe DRSM-L, show the optimal value of its principal design parameter, and present results indicating the performance under simulation.

Key Words and Phrases: algorithm, checkpoint, fault tolerance, hardware, multiprocessor

I. Introduction

The tightly-coupled multiprocessor (TCMP), where specialized hardware maintains the image of a single shared memory, has the highest performance among the various types of computer systems. In order to facilitate the use of such TCMPs in the commercial environment, we must build fault tolerance into them. The dominant method of fault tolerance is roll-back recovery. It has 2 principal aspects. First, a processor establishes occasional checkpoints; a checkpoint is a consistent state of the system. Second, if the processor encounters a fault, the processor rolls back to the last checkpoint and commences execution from the state saved in that checkpoint. The first aspect, the establishment of checkpoints, is the more important one as it is a cost that the TCMP regularly experiences even if no fault arises. The second aspect, the actual rolling-back, is less important as faults tend to occur infrequently. Hence, much of the research in roll-back recovery for TCMPs has focused on developing efficient algorithms for establishing checkpoints.

In this paper, we present the first apparatus and algorithm enabling a processor to perform roll-back or checkpoint establishment in a way that is independent of any other processor in a TCMP. Our algorithm is called distributed recoverable shared memory nith logs (DRSM-L). In section II, we discuss the basic issue of dependency. In section III, we present our assumptions. We then describe DRSM-L in section IV. In Section V, we discuss the principal design parameter of DRSM-L. In section VI, we describe our simulation environment and methodology. In section VII, we present our experimental results. We conclude the paper in section VIII.

II. Background

A. Dependencies

The basic idea of roil-back recovery is the following. In a uni-processor computer, the processor periodically establishes a checkpoint, a consistent state of the system. If it encounters a fault, the processor rolls the system back to the state in the last checkpoint.

This simple scheme becomes complicated in a TCMP. Processors access shared memory blocks, and this interaction causes dependencies to arise. There are 4 possible types of interactions on the same shared memory block.

1. **read – read**: A read by processor P precedes a read by processor Q.
   dependency: none
2. **read – write**: A read by processor P precedes a write by processor Q.
   dependency: none
3. **write – read**: A write by processor P precedes a read by processor Q.
   roll-back dependency: P -> Q
   checkpoint dependency: Q -> P
4. **write – write**: A write by processor P precedes a write by processor Q.
   roll-back dependency: P <-> Q
   checkpoint dependency: P <-> Q

Only the last 2 interactions cause dependencies to arise. We shall examine how they arise. In our presentation, we assume that a memory block and the highest-level-cache line are identical in size and that the TCMP uses a write-back cache policy. To minimize the cost of the system, we assume that it can hold only 1 level of checkpoint.

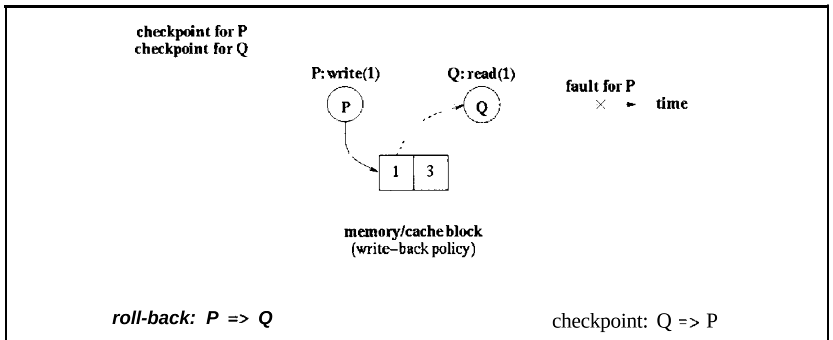


Figure 1. Roll-back Dependency for Write-Read Interaction

Figure 1 illustrates the roll-back dependency that arises for the write-read interaction. After processor "P" writes the value of 1 into a word of the memory block, processor "Q" reads that value of 1. Then, processor "P" experiences a fault and rolls back to the last checkpoint. "Q" must also roll back to the last checkpoint because "Q" read a value, l in this case, that "P" produced. When "P" resumes execution from the last checkpoint, "P" may not necessarily reproduce the value of 1. Thus, we have the roll-back dependency of "P -> Q".

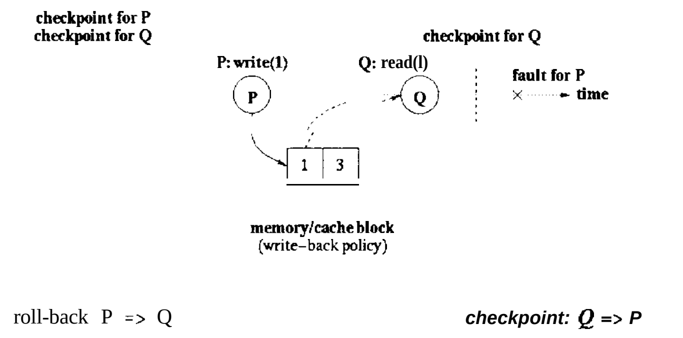
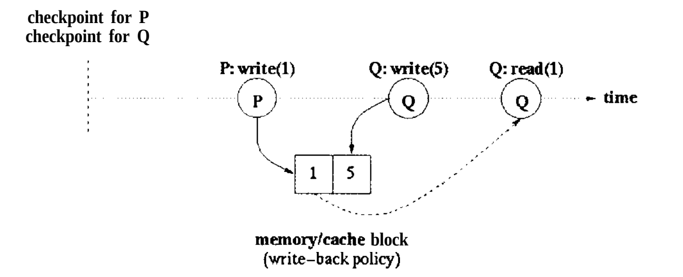


Figure 2. Checkpoint Dependency for Write-Read Interaction

Figure 2 illustrates the checkpoint dependency that arises for the write-read interaction. After processor "P" writes the value of 1 into a word of the memory block, processor "Q" reads that value of I. Then, processor "Q" establishes a checkpoint. In figure 2, "Q" establishes 2 checkpoints: the 1st checkpoint occurring before "Q" reads the value of 1 and the 2nd checkpoint occurring after "Q" reads the value of 1. Subsequently, processor "P" experiences a fault and rolls back to the last checkpoint. The roll-back dependency dictates that "Q" must roll back to the 1st checkpoint, but by the time that "P" experiences a fault, "Q" has already established the 2nd checkpoint and can only roll back to it. The state saved in the 2nd checkpoint depends on the value of l produced by "P". Since "P" may not necessarily reproduce the value of 1 after resuming execution from the last checkpoint, the state saved in the 2nd checkpoint can be invalid. Therefore, if "Q" establishes a checkpoint after the write-read interaction, then "P" must also establish a checkpoint. In this way, we ensure that "P" does not "m-do" the value of l that was read by "Q". Thus, we have the checkpoint dependency of "Q -> P".

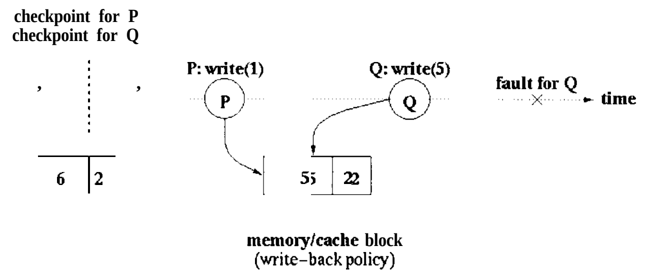
The write-write interaction has 2 cases: one where the processors write into different words of the same memory block and one where the processors write into the same word of the same memory block. Each case results in a different direction of the dependency (i. e. a different direction of the dependency arrow in the above list). The combined effect of both cases is a 2-way dependency for both the roll-back dependency ("P <-> Q") and the checkpoint dependency ("P <-> Q").



roll-back: P => Q checkpoint: Q => P

Figure 3. Roll-back/Checkpoint Dependency for Write-Write Interaction with Writes to Different Words

Figure 3 illustrates the roll-back dependency and the checkpoint dependency for the write-write interaction where the processors write into different words of the same memory block. Since the cache-policy is write-back, after "Q" writes the value of 5 into the block, "Q" holds it in the cache. "Q" can potentially read the value of 1 written by “P”. Hence, a write-read interaction arises. We have already analyzed this interaction in figures 1 and 2. Thus, we have a roll-back dependency of "P -> Q" and a checkpoint dependency of "Q -> P”.



roll-back: Q => P checkpoint: P => Q

Figure 4. Roll-back Dependency for Write-Write Interaction with Writes to the Same Word

Figure 4 illustrates the roll-back dependency for the write-write interaction where the processors write into the same word of the same memory block. Processor "P" writes the value of 1 into the memory block, overwriting the original value of “6”. Then, processor "Q" writes the value of 5 into the same word, overwriting the value of 1. Subsequently, "Q" experiences a fault and rolls back to the last checkpoint. "Q" undoes the value of 5 and must replace it with the value of 1, but there is no convenient way to retrieve the value of 1 since it was destroyed by "Q" writing the value of 5. The only value that the TCMP can use to replace 5 is the original value of 6. In order to ensure that the state of the TCMP is valid, "P" must roll-back to the last checkpoint as well in order to un-do the value of 1. Hence, we have a roll-back dependency of "Q -> P”.

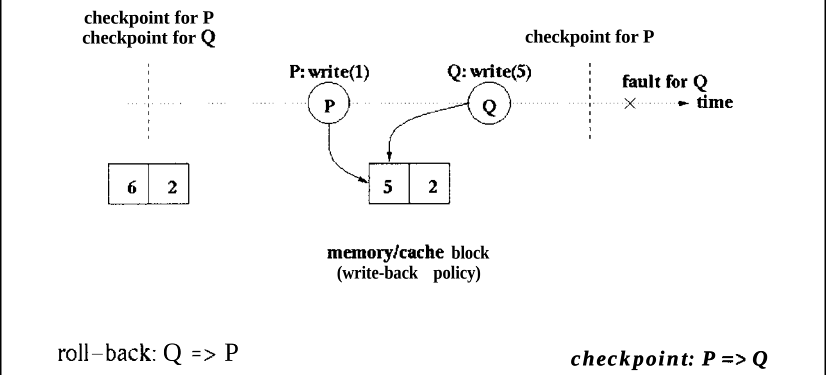


Figure 5. Checkpoint Dependency for Write-Write Interaction with Writes to the Same Word

Figure 5 illustrates the checkpoint dependency for the write-write interaction where the processors write into the same word of the same memory block. Processor "P" writes the value of 1 into the memory block, overwriting the original value of “6”. Then, processor "Q" writes the value of 5 into the same word, overwriting the value of 1. Next, "P" establishes a checkpoint. In figure 5, "P" establishes 2 checkpoints: the 1st checkpoint occurring before "P" writes the value of 1 and the 2nd checkpoint occurring after "P" writes the value of 1. Subsequently, "Q" experiences a fault and rolls back to the last checkpoint. The roll-back dependency dictates that "P" must roll back to the 1st checkpoint, but by the time that "Q" experiences a fault, "P" has already established the 2nd checkpoint and can only roll back to it. The state of the TCMP can become invalid since (1) it assumes that the value of 1 is stored in the pertinent word of the memory block but (2) "Q" can un-do the value of 5 by only replacing it with 6. There is no convenient way to retrieve the value of 1 and to use I to replace 5. Hence, to solve this problem, if "P" establishes the 2nd checkpoint, "Q" must also establish a checkpoint. Then, "Q" will not roll back past the 2nd checkpoint and will not need to un-do the value of 5. Thus, we have a checkpoint dependency of "P -> Q"

Therefore, the write-write interaction causes bi-directional dependencies to arise. The roll-back dependency is "P <-> Q”. The checkpoint dependency is "P <-> Q" as well.

B. Classes of Algorithms

Checkpointing algorithms must deal with these dependencies. How the algorithms deal with them results in a natural partition of the types of algorithms that exist.

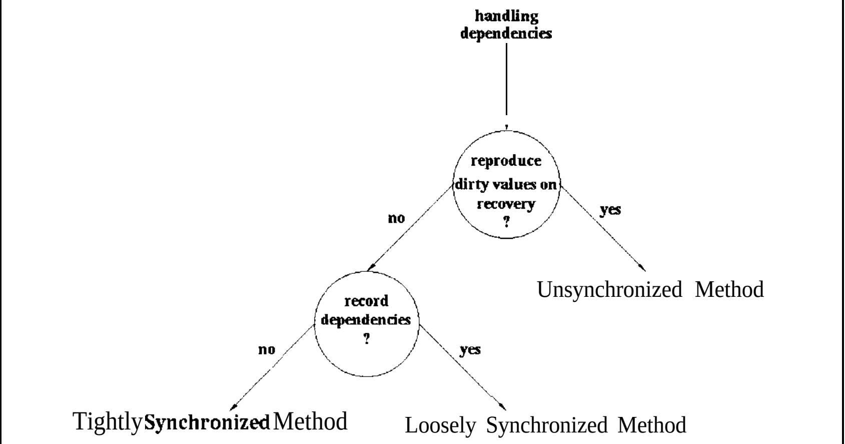

handling dependencies
reproduce dirty values on recovery ?
no
yes
Unsynchronized Method
record dependencies ?
no
yes
Tightly Synchronized Method
Loosely Synchronized Method

Figure 6. Dependencies to Classes of Checkpointing Algorithms

Figure 6 illustrates the approaches for dealing with the dependencies. Dependencies arise because a dirty value written by a processor "P" (and possibly read by another processor "Q") is not necessarily reproduced if "P" rolls back to its last checkpoint and resumes execution from it. If an algorithm can ensure that dirty values written by a processor "P" after roll-back are identical to those dirty values written by "P" before encountering the fault (that resulted in the roll-back), then the algorithm is a member of the class called the unsynchronized method (USM). In a USM-type algorithm, a processor can establish a checkpoint (or perform a roll-back) that is completely independent of any other processor.

If an algorithm cannot guarantee that dirty values produced by "P" after roll-back are identical to those dirty values produced by "P" before encountering the fault, then the algorithm can simply record the dependencies. Such an algorithm is a member of the class called the loosely synchronized method (LSM). In a LSM-type algorithm, if any checkpoint dependency (or roll-back dependency) arises, the TCMP simply records the dependency. At some point in the future, if a processor establishes a checkpoint (or performs a roll-back), then that processor queries the records of dependencies to determine all other processors that must establish a checkpoint (or perform a roll-back) as well.

Finally. if the checkpointing algorithm cannot guarantee reproduction of dirty values after roll-back and if the algorithm does not record dependencies, then the algorithm must do following. A processor "P" must establish a checkpoint whenever "P" delivers dirty data to another processor or to the memory system (via the eviction of a dirty cache line). Such an algorithm is a member of the class called the tightly synchronized method (TSM).

For a TCMP, researchers have developed algorithms in 2 classes, TSM and LSM. An example of a TSM-type algorithm is the one proposed by Wu [13]. An example of a LSM-type algorithm is the one proposed by Banatre [2]. In this paper, we present DRSM-L, the first USM-type algorithm for a TCMP. Before DRSM-L, no such algorithm for a TCMP existed although Richard [6] and Suri [11] have proposed USM-type algorithms for a loosely-coupled multiprocessor like a network of workstations. The DRSM-L described in this report is an improved version of DRSM-L originally contrived by Sunada [10].

III. Assumptions

The TCMP into which we shall incorporate DRSM-L is a multi-node multiprocessor like that shown in Figure 7. Each node has a processor module and a memory module. The nodes are connected by a high-speed dedicated network. We make the following specific assumptions about our TCMP.

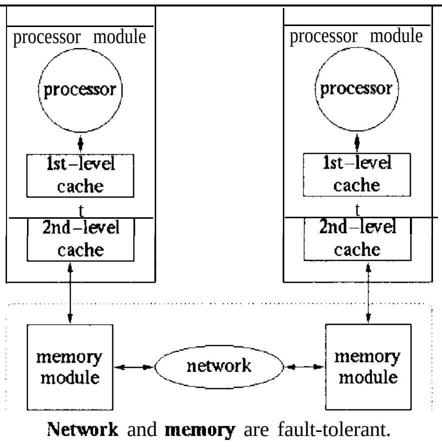


Figure 7. Tightly-Coupled Multiprocessor (TCMP)

1. Each component in our TCMP is fail safe. If the component fails, it simply stops and does not emit spurious data.
2. The TCMP suffers at most a single point of failure.
3. Each memory module is fault tolerant.
4. The network connecting the nodes in our TCMP is fault tolerant. Specifically, between any 2 nodes are 2 independent paths, and each memory module is dual-ported.
5. The virtual machine monitor (VMM) is fault-tolerance aware. Specifically, if communication occurs between a processor and the environment outside of the TCMP, then the VMM will invoke the processor to establish a checkpoint.

The first 4 assumptions are commonly found in research papers proposing checkpointing algorithms for a TCMP. The last assumption can be re-phrased by replacing VMM with operating system (OS) for those systems without a VMM. We believe that building fault tolerance into the VMM is superior to building fault tolerance into the OS since a fault-tolerant VMM enables us to run any non-fault-tolerant OS while the entire TCMP remains fault-tolerant. The TCMP views the OS as simply another user application running on top of the VMM [3].

The gist of our assumptions is that the memory module and network are fault tolerant but that the processor module (including the processor and associated caches) are not fault tolerant. DRSM-L is an apparatus and algorithm that enables the TCMP to recover from a failure of the processor module.

IV. Distributed Recoverable Shared Memory with Logs (DRSM-L)

Distributed recoverable shared memory with logs (DRSM-L) is a USM-type algorithm and apparatus to establish checkpoints for fault tolerance. DRSM-L enables a processor to establish a checkpoint (or to roll back to the last checkpoint) in a manner that is independent of any other processor.

A. Apparatus

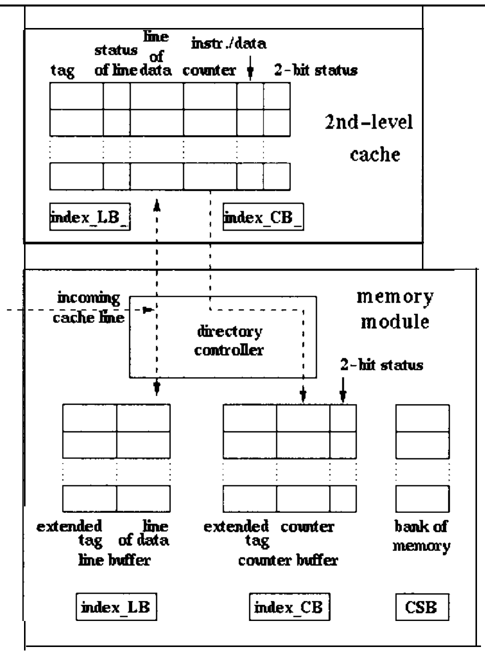


Figure 8. Distributed Recoverable Shared Memory with Logs (DRSM-L)

Figure 8 illustrates the apparatus of DRSM-L. It consists of new structures in both the 2nd-level cache and the local memory module. Each line of the 2nd-level cache has the traditional fields: tag, status (SHARED, EXCLUSIVE, and INVALID) of line, and line of data. Each line has 3 additional fields: counter, instruction/data flag (IDF), and 2-bit status flag (SF). The SF assumes any 1 of 4 values: “N” (no event), “R” (remote read), “E” (ejection), “V” (counter overflow). The cache also has 2 index registers that mirror 2 index registers in the local memory module.

The local memory module has the traditional directory controller and bank of memory. The module also has a line buffer and a counter buffer. Each buffer has an accompanying index register that points at the next free entry in the buffer. The module also has a checkpoint-state buffer (CSB). We will describe how the new structures function as we describe how DRSM-L logs incoming cache data, how DRSM-L establishes checkpoints, what triggers the establishment of checkpoints, and how DRSM-L facilitates recovery from a fault.

In the following discussion, we assume that the TCMP (1) prohibits self-modifying code and (2) requires instructions and normal data to reside in separate memory blocks (i. e. cache lines). In section IV-C, we explain how this assumption allows us to use the IDF.

B. Audit Trail

DRSM-L is a IJSM-type algorithm and guarantees that a processor resuming execution from the last checkpoint (after a roll-back due to a fault) reproduces the exact same dirty values that the processor

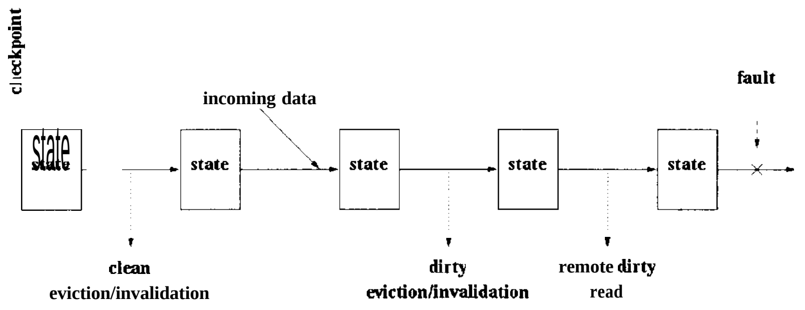


Figure 9. Transition of State of Both Processor and 2nd-Level Cache

produced before encountering a fault. Figure 9 illustrates the strategy that DRSM-L uses to provide this guarantee. In figure 9, the “state” is the combination of the processor state and the 2nd-level-cache state; the cache state refers only to 3 fields -- tag, status of line, line of data, and the IDF. The transition of the processor-cache state depends on 4 events: incoming data in the form of memory blocks, clean evictions (or invalidations), dirty evictions (or invalidations), and dirty reads by a remote processor. During recovery after a fault, the processor will reproduce the dirty values delivered by the dirty eviction (or invalidation) and by the remote dirty read if the processor-cache state repeats the transitions (that occurred prior to the fault) past the point where the processor sent the last dirty value to the rest of the TCMP. In figure 9, the last dirty value sent by the processor occurs during the remote dirty read. Furthermore, the processor-cache state will repeat the transitions (that occurred prior to the fault) if the recovery apparatus reproduces the 4 events at the right time relative to the last checkpoint and to the data accesses. Therefore, to ensure that a processor "P" can reproduce the exact same dirty values that it produced prior to a fault, "P" must record (1) data values that arrive in its 2nd-level cache and (2) the number of times that "P" accesses each 2nd-level-cache line of data until the next event that occurs on the cache line. Events that occur on a cache line are clean eviction/invalidation, dirty eviction/invalidation, and remote dirty read.

"P" logs the incoming cache lines (or memory blocks) into the line buffer in figure 8. The line buffer has 2 fields: extended tag and line of data. The extended tag is the regular 2nd-level-cache tag appended with the index of the exact cache line into which the incoming cache data is destined. The "line of data" is the cache data itself. The logging of the incoming cache line into the line buffer can be performed in parallel with forwarding the line to the 2nd-level cache, so <u>the logging does not cause additional delay</u>.

Merely recording the incoming cache line is not sufficient to guarantee that "P" will reproduce the exact same dirty values that it produced prior to a fault. "P" must also record the number of times that "P" uses data in a 2nd-level-cache line before an event occurs on it. Below are the 3 possible events.

1. The status of the line transitions from EXCLUSIVE to SHARED due to a read by a remote processor.
2. The status of the line transitions from SHARED or EXCLUSIVE to INVALID due to an eviction or invalidation.
3. The counter overflows.

To count the number of accesses to data in a 2nd-level cache prior to these events, "P" performs the following. "P" forwards the address of the access to both the 1St-level cache and the 2nd-level cache. Regardless of whether the access hits in the 1st-level cache, if the access hits in a line of the 2nd-level cache, it increments the counter of the matching line. The counter of a line is reset to 0 whenever incoming cache data arrives in the line.

If a valid 2nd-level-cache line experiences any 1 of the above 3 events, then the cache writes the counter of that cache line into the counter buffer in figure 8 and resets the counter (in the cache line) to 0. The counter buffer has 3 fields: extended tag, counter, and 2-bit status flag (SF). The directory controller sets the SF to "R", "E", or

"V" if event #1, #2, or #3, respectively, occurs. For events #1 and #2, logging the counter into the counter buffer is performed in parallel with the usual cache-coherence activity, so the logging does not cause additional delay.

The line buffer and counter buffer effectively record an audit trail of data accessed by a processor "P". After "P" encounters a fault, "P" rolls back to the last checkpoint and resumes execution. "P" uses the audit trail to satisfy all read or write misses until recovery is complete. The information stored in the line buffer and the counter buffer is sufficient and necessary to ensure that "P" will reproduce the exact same dirty values that it produced prior to a fault. We illustrate how recovery works in section IV-F.

If an upgrade miss (i. e. a write "hit" on a 2nd-level-cache line with its state being SHARED) occurs in the 2nd-level-cache, it obtains permission to upgrade the affected cache line in the same way that the 2nd-level cache of a base TCMP without DRSM-L would handle the upgrade miss. Upgrading a 2nd-level-cache line involves changing its status from SHARED to EXCLUSIVE. After the 2nd-level cache upgrades the affected cache line, the cache retries the data-write. The cache then processes the write hit in the usual fashion for maintaining an audit trail.

C. Optimizations

For the 2nd-level cache, figure 8 shows 2 optimizations: the instruction/data flag (IDF) and the index registers which mirror those in the local memory module. If the TCMP both (1) prohibits self-modifying code and (2) requires instructions and normal data to reside in separate memory blocks (i. e. cache lines), then we can distinguish between cache lines holding instructions and cache lines holding data. If the incoming cache line satisfies an access miss for regular data, then the 2nd-level cache sets the IDF of the cache line (satisfying the access miss) to 1. The incoming cache line and its associated counter are handled in the usual manner for maintaining an audit trail. On the other hand, if the incoming cache line satisfies an instruction miss, then the cache line is not saved in the line buffer. Further, the 2nd-level cache sets the IDF of the cache line (satisfying the instruction miss) to 0. Subsequent instruction fetches that hit in the cache line do not cause the counter to increment. In this way, we avoid using space in the line buffer and the counter buffer to store the instructions and the number of instruction fetches, respectively.

If the TCMP either (1) allows self-modifying code or (2) allows instructions and normal data to reside in the same memory block (i. e. cache line), then we cannot distinguish between cache lines holding instructions and cache lines holding data. We omit the IDF flag and must handle all incoming cache lines in the usual manner for maintaining an audit trail.

The second optimization is the index registers that mirror those in the local memory module. In the memory module, the index register of each buffer points to the next free entry in the buffer. “index-LB” is the index register of the line buffer, and “index_CB” is the index register of the counter buffer. “index_LB_” and “index_CB_” mirror “index-LB” and “index_CB”, respectively. When the directory controller saves an incoming cache line (for a data access) into the line buffer and forwards the line to the 2nd-level cache, the directory controller increments “index-LB”. The 2nd-level cache installs the incoming cache data into a cache line and increments “index_LB_”. In addition, when the 2nd-level cache writes a counter into the counter buffer, the cache increments “index_CB_”. When the directory controller saves the counter into the counter buffer, the directory controller increments “index_CB”.

When the line buffer or the counter buffer becomes full, the local processor must establish a checkpoint, Establishing a checkpoint clears both buffers and resets all the index registers to 0. The index registers in the 2nd-level cache itself enable the processor module to determine whether the line buffer or the counter buffer is full without incurring the cost of querying the index registers in the local memory module.

D. Checkpoint Establishment

The checkpoint-state buffer (CSB) assists the local processor to establish a checkpoint. The CSB has 3 separate units (which are not shown in figure 8): checkpoint flag (CF), tentative-checkpoint area, and the permanent-checkpoint area. The CF indicates 1 of 3 checkpointing phases: "CHECKPOINT_IS_NOT_ACTIVE", "TENTATIVE_CHECKPOINT_IS_ACTIVE", and "PERMANENT_CHECKPOINT_IS_ACTIVE". The tentative-checkpoint area holds both (1) the processor state (of the local processor) and (2) the contents of the 2nd-level cache for the current checkpoint. The permanent-checkpoint area holds both (1) the processor state and (2) the contents of the 2nd-level cache from the last checkpoint. At the end of the establishment of the current checkpoint, the local processor switches the designation of the tentative-checkpoint area and the permanent-checkpoint area. In other words, the tentative-checkpoint area becomes the permanent-checkpoint area, and the permanent-checkpoint area becomes the tentative-checkpoint area.

DRSM-L enables a processor "P" to establish a checkpoint in 2 phases: tentative checkpoint and permanent checkpoint. "P" first updates the CF of the CSB to "TENTATIVE_CHECKPOINT_IS_ACTIVE", indicating that "P" is in phase 1, the tentative checkpoint. Then, "P" waits until all its pending memory accesses are complete (or negatively acknowledged). "P" negatively acknowledges all cache-coherence messages from the directory controllers until the establishment of the checkpoint is complete. Next, "P" downloads both its internal registers (i. e. the processor state) and all 2nd-level-cache lines (saving only the tag, status of line, line of data, and IDF) into the tentative-checkpoint area of the CSB (while preserving the previous permanent checkpoint in the permanent-checkpoint area). At the end of phase 1, "P" updates the CF of the CSB to "PERMANENT_CHECKPOINT_IS_ACTIVE", indicating that "P" is now in phase 2 (i. e. the permanent checkpoint). "P" resets all the index registers in both the processor module and the local memory module. "P" then tells the CSB to invalidate the old permanent checkpoint in the CSB and to designate the processor state and cache lines saved in the tentative-checkpoint area as the nen permanent checkpoint. Finally, "P" updates the CF of the CSB to "CHECKPOINT_IS_NOT_ACTIVE", indicating that phase 2 (and the entire checkpoint) is finished.

E. Triggers of Checkpoint Establishment

Any 1 of the following 4 conditions can trigger the establishment of a checkpoint.

1. A timer expires. The timer determines the maximum temporal interval between checkpoints.
2. The line buffer overflows.
3. The counter buffer overflows.
4. Communication occurs between the processor and the environment outside of the TCMP.

F. Recovery from a Fault

Fault-tolerance schemes generally involve 2 aspects: (1) logging data or establishing periodic checkpoints prior to the occurrence of any fault and (2) rolling the system back to the last checkpoint and recovering the state of the system prior to the fault. We now show how DRSM-L may recover from a fault. We assume that the TCMP operates with a fault-tolerance-aware virtual machine monitor (VMM). (The following comments apply as well to a TCMP that executes a fault-tolerance-aware operating system without a VMM). We consider the following simple scheme for rolling back from a fault experienced by a processor. We arrange for a special recovery logic circuit (RLC) on the memory module to periodically send "Are you alive?" messages to the local processor. If it does not respond within a specified timeout period, RLC assumes that the processor experienced a fault. If the fault is permanent, RLC replaces the failed processor with a spare processor. Then, RLC resets the processor, say "P", and directs it to begin the recovery activity. "P" invalidates all entries in both the Ist-level cache and the 2nd-level cache. "P" negatively acknowledges all cache-coherence messages from the directory controllers until recovery is complete.

If "P" failed during the establishment of the permanent checkpoint, according to the CF of the CSB, then "P" completes the establishment of the permanent checkpoint that was in progress when the fault occurred. "P" queries all the memory modules to find messages which were sent to "P" just prior to the fault; "P" negatively acknowledges them. Then, "P" loads both the processor state and all 2nd-level-cache lines saved in the permanent-checkpoint area of the CSB. "P" installs each line saved in the CSB into a 2nd-level-cache line and resets its counter to 0. "P" resumes normal processing.

If "P" did not fail during the establishment of the permanent checkpoint, then "P" must perform the following procedure. "P" queries all the memory modules to find messages which were sent to "P" just prior to the fault; "P" negatively acknowledges them. Then, "P" reads the entire line buffer and the entire counter buffer and groups their entries according to the cache index of the extended tag so that an entry can be easily fetched on a miss in the 2nd-level cache. "P" saves these sorted entries in a separate memory area reserved for the VMM; for the purpose of this discussion, we assume that they reside in the sorted line buffer (SLB) and the sorted counter buffer (SCB). Then, "P" invalidates all entries in both the lst-level cache and the 2nd-level cache and loads both the processor state and all 2nd-level-cache lines saved in the permanent-checkpoint area of the CSB. "P" installs each line saved in the CSB into a 2nd-level-cache line, resets its counter to 0, and sets SF to "V".

"P" resumes execution in recovery mode. In this mode, if a data-read or data-write misses in the 2nd-level cache, a trap occurs to the VMM. It finds the next matching line (of data) and counter from the SLB and SCB, respectively, and places the line and counter into the cache. The VMM sets the IDF of the affected cache line to "1". The VMM also sets the SF in the cache to the value stored in the SF of the SCB.

If a data-read or a data-write hits on a 2nd-level-cache line, its counter is decremented. Once a hit causes the counter to underflow (below 0), the hardware must interpret the SF. If the SF is "N", then the counter is not decremented but remains at 0, and satisfying the access hit proceeds. On the other hand, if the SF is not "N", then execution traps to the VMM. It must interpret the SF. If it is "R", then the VMM re-loads the counter and SF with the next matching entry in the SCB and also transitions the status of the line from EXCLUSIVE to SHARED. If the SF is "E", then the VMM re-loads the cache line with the next matching entry in the SLB and also re-loads the counter and SF with the next matching entry in the SCB. If the SF is "V", then the VMM re-loads the counter and SF with the next matching entry in the SCB. Regardless of whether the SF is "R", "E", or "V", if the next matching entry in the SCB does not exist, then the VMM sets the counter and the SF to 0 and "N", respectively.

Also, if an upgrade miss (i. e. a write "hit" on a 2nd-level-cache line with its status being SHARED) occurs in the 2nd-level cache, it immediately upgrades the affected cache line by changing its state from SHARED to EXCLIJSIVE. The 2nd-level cache then retries the data-write and processes it in the usual fashion for recovery. We specifically note that the 2nd-level cache does not handle the upgrade miss by submitting a cache-coherence message to the directory controller.

In the recovery mode, "P" handles instruction misses by fetching the instruction from main memory into the cache in the usual fashion. "P" sets the IDF (of the 2nd-level-cache line receiving the incoming memory block that satisfies the instruction miss) to "0". For data misses (due to data-reads and data-writes), the processor sets the IDF to "1" but uses the SLB and the SCB to satisfy them.

Eventually, "P" achieves the following 2 conditions: (1) the SCB has no entry with a SF of "E" or "R" and (2) the counters in all valid cache data lines (i. e. with IDF being "1") with a SF of "E" or "R" are 0. When both conditions arise, recovery for "P" is close to completion. We say that "P" has reached the state of "imminent completion of recovery". Execution traps to the VMM. It updates the status of all valid cache lines with a SF of "E" or "R". If the SF is "E" or "R", then "P" changes the status of the line to INVALID or SHARED, respectively. Then, the VMM invokes "P" to establish a checkpoint, which clears both the line buffer and the counter buffer.

To ensure that the contents of the cache are consistent with the information stored in the directory of each memory module, the VMM reads each dirty 2nd-level-cache line from the permanent-checkpoint area of the CSB and writes the line back into main memory. The VMM changes the status of each 2nd-level-cache line saved in the permanent-checkpoint area (of the CSB) to INVALID. Then, the VMM tells the directory controller of each memory module to change the status of any memory block (i. e. cache line) held by the 2nd-level cache (of "P") to indicate that the 2nd-level cache no longer holds the memory block. The VMM also changes the status of each line in the 2nd-level cache to INVALID.

Finally, recovery is complete. The VMM then places "P" in the normal mode of execution where the counter increments on each hit.

We note that DRSM-L has the extremely desirable property of no roll-back propagation. If a processor experiences a fault, the processor (or the spare processor) must roll back to the last checkpoint to resume execution. This roll-back does not require that other processors also roll back to their last checkpoints.

G. Precise Description of DRSM-L

Below, we use C-like code to precisely describe how the various pieces of DRSM-L work. For the sake of brevity, we label the 2nd-level cache as simply "cache".

Also, we try to restrict our description to only those activities that occur in a TCMP with DRSM-L but that do not necessarily occur in a base TCMP without DRSM-L. For example, we do not describe the updating of the fields in the 2nd-level-cache line when an upgrade miss (i. e. write "hit" on a line with status being SHARED) is processed.

Finally, we do not explicitly identify the activities that can occur in parallel for increased performance. The parallel activities should be obvious from the context. For example, when the cache installs an incoming memory block (satisfying a data access) into a 2nd-level-cache line, both (1) resetting the counter to 0 and (2) setting the IDF to 1 can occur in parallel.

```
explanatory notes
-----------------
States of cache line are INVALID, SHARED, and EXCLUSIVE.
Number of entries in line buffer is 8192 (which can be changed).
Number of entries in counter buffer is 8192 (which can be changed).
Width of counter is 32 bits (which can be changed).



execution mode:  normal
-----------------------
switch (event) {

   /*--------------------------------------------------------------------*/
   /*   Upgrade miss occurs in cache data line.                          */
   /*                                                                    */
   /*   Upgrade miss occurs when data-write "hits" in cache              */
   /*        line with status being SHARED.                              */
   /*--------------------------------------------------------------------*/

   data-write-has-upgrade-miss-in-cache-data-line:  {

      /*-----------------------------------------------------------------*/
      /*   Data-write stalls until "cache_line.status of line"           */
      /*        becomes EXCLUSIVE in response to upgrade-miss.           */
      /*                                                                 */
      /*   After local processor retries stalled data-write,             */
      /*        it then will hit in cache data line and will             */
      /*        generate "data access-hits-in-cache-data-line"           */
      /*        as next event.-                                          */
      /*-----------------------------------------------------------------*/



      break;
   }

   /*--------------------------------------------------------------------*/
   /*   Access misses in cache data line.                                */
   /*--------------------------------------------------------------------*/

   data_access_misses_in_cache_data_line: {
      if (index_LB_ == 0x2000) {

         /*--------------------------------------------------------------*/
         /*   8192-entry line buffer is full, so empty it.               */
         /*--------------------------------------------------------------*/

         stall data access;
         establish_checkpoint();
         retry data access;
      }
      break;
   }

   memory-block-arrives:  {
      if (original access is data access) {
        log-memory-block-into line_buffer();
        install_memory_block_into_cache_data_line();
    }
    else {
        install_memory_block_into_cache_instruction_line();
    }
    break;
}

/*-------------------------------------------------------------------*/
/*   Access hits in cache line.                                       */
/*-------------------------------------------------------------------*/

data_access_hits_in_cache_data-line: {
    if (cache_line.counter == 0x0FFFFFFFF) {

        /*------------------------------------------------------------*/
        /*   32-bit counter overflows.                                */
        /*------------------------------------------------------------*/

        stall data access;
        if (index_CB_ == 0x2000) {

            /*---------------------------------------------------------*/
            /*   8192-entry counter buffer is full, so empty it.       */
            /*---------------------------------------------------------*/

            establish_checkpoint();
        }
        else {
            index_CB_++;
            log_counter_into_counter_buffer(cache_line, "V");
        }
        retry data access;
    }
    else {
        cache_line.counter++;
    }
    break;
}

/*-------------------------------------------------------------------*/
/*   Eviction or invalidation of cache line occurs.                   */
/*-------------------------------------------------------------------*/

evict cache line:
invalidate_cache_line: {
    if (index_CB_ == 0x2000) {

        /*------------------------------------------------------------*/
        /*   8192-entry counter buffer is full, so empty it.          */
        /*------------------------------------------------------------*/

        stall event, which is "evict-cache-line" or "invalidate_cache_line";
        establish_checkpoint();
        retry event, which is "evict-cache-line" or "invalidate_cache_line";
    }
    else {
        /*-------------------------------------------------------------*/
        /*   No additional delay is incurred.                          */
        /*-------------------------------------------------------------*/

        index_CB_++;
        log_counter_into_counter_buffer(cache_line, "E");
      }
      break;
   }

   /*-------------------------------------------------------------------*/
   /*   Remote processor reads local dirty cache line.                  */
   /*-------------------------------------------------------------------*/

   remotely read-local-dirty-cache-line: {
      if (index_CB_ == 0x2000) {

        /*-------------------------------------------------------------*/
        /*   8192-entry counter buffer is full, so empty it.           */
        /*                                                             */
        /*   Write-back is response by local processor to remote       */
        /*        processor reading local dirty cache line.            */
        /*-------------------------------------------------------------*/

        stall write-back;
        establish_checkpoint();
        retry write-back;
      }
      else {

        /*-------------------------------------------------------------*/
        /*   No additional delay is incurred.                          */
        /*-------------------------------------------------------------*/

        index_CB_++;
        log_counter_into_counter_buffer(cache_line, "R");
      }
      break;
   }

   /*-------------------------------------------------------------------*/
   /*   Timer expires, or I/O occurs.                                   */
   /*-------------------------------------------------------------------*/

   timer-expires:
   communication_between_cpu_and_environment_outside_TCMP_occurs: {
      establish_checkpoint();
      break;
   }

   default: {
      do nothing special;
   }
}

log_memory_block_into_line_buffer()
{
   /*-------------------------------------------------------------------*/
   /*   Update local memory module.                                     */
   /*-------------------------------------------------------------------*/

   line_buffer[index_LB].extended_tag <= extended-tag(memory-block);
   line_buffer[index_LB].line_of_data <= data(memory_block);

   index_LB++;
}

install_memory_block_into_cache_data_line()
{

   /*-------------------------------------------------------------------*/
   /*   Update cache.                                                   */
   /*-------------------------------------------------------------------*/

   index_LB_++;

   /*-------------------------------------------------------------------*/
   /*   Update cache line.                                              */
   /*-------------------------------------------------------------------*/

   cache-line.tag <= tag(memory_block);
   if (data access == read) {
      cache_line.status_of_line <= SHARED;
   }
   else {
      cache_line.status_of_line <= EXCLUSIVE;
   }
   cache_line.line_of_data <= data(memory_block);

   cache_line-counter <= 0;
   cache line.IDF <= 1;
   cache-line.SF <= "N";
}

available_cache_line()
{
   /*-------------------------------------------------------------------*/
   /*   Get cache line (from set of set-associative cache) to           */
   /*        hold incoming data.                                        */
   /*-------------------------------------------------------------------*/

   if (set has cache-line where cache-line.status_of-line == INVALID) {
      get cache-line where cache_line.status_of_line == INVALID;
   }
   else if (set has cache_line where cache line.IDF == 1 and
                                     cache-line.counter == 0 and
                                     cache-line.SF == "E") {
      get cache-line where cache-line.IDF == 1 and
                           cache line.counter == 0 and
                           cache-line.SF == "E";
   }
   else if (set has cache line where
                 cache-line.IDF == 1 and
                 cache-line.counter == 0 and
```

```
                cache line.SF == "V" and
                sorted-counter-buffer[entry].counter == 0 and
                sorted-counter-buffer[entry].SF == "E" and
                extended_tag(cache_line) ==
                   sorted counter_buffer[entry].extended_tag
            for next matching entry in sorted counter buffer) {

      /*------------------------------------------------------------------*/
      /*   Set can have cache-line where                                 */
      /*                                                                  */
      /*        cache line.IDF == 1,                                      */
      /*        cache-line-counter == 0,                                  */
      /*        cache-line.SF == "V",                                     */
      /*        sorted-counter-buffer[entry].counter == 0,                */
      /*        sorted-counter-buffer[entry].SF == "E", and               */
      /*        extended_tag(cache_line) ==                               */
      /*           sorted-counter-buffer[entry].extended_tag              */
      /*                                                                  */
     /*    for next matching entry in sorted counter buffer.              */
      /*                                                                  */
      /*   Situation can arise, for example, if immediately after        */
      /*        establishing checkpoint, processor resumes normal         */
      /*        execution and issues data access that experiences         */
      /*        conflict miss, which evicts cache line.                   */
      /*------------------------------------------------------------------*/

      get cache-line where
          cache_line.IDF == 1 and
          cache_line-counter == 0 and
          cache line.SF == "V" and
          sorted-counter-buffer[entry].counter == 0 and
          sorted-counter-buffer[entry].SF == "E" and
          extended_tag(cache_line) ==
             sorted_counter_buffer[entry].extended_tag
         for next matching entry in sorted counter buffer;

      /*------------------------------------------------------------------*/
      /*   Retrieve and discard matching entry mentioned                  */
      /*        in directly preceding statement.                          */
      /*------------------------------------------------------------------*/

      retrieve and discard next entry from sorted counter buffer,
         which matches cache_line;
   }
   else {
      get cache-line where cache-line.IDF == 0;
   }

   return cache_line;
}

install_memory_block_into_cache_instruction_line()
{

   /*---------------------------------------------------------------------*/
   /*   Update cache line.                                                */
   /*---------------------------------------------------------------------*/
   cache-line-tag <= tag(memory_block);
   cache_line.status_of_line <= SHARED;
   cache_line.line_of_data <= data(memory_block);

   cache line.counter <= 0;
   cache-line.IDF <= 0;
   cache-line.SF <= "N";
}

log_counter_into_counter_buffer(cache_line, event)
{

   /*---------------------------------------------------------------*/
   /*   Update local memory module.                                 */
   /*---------------------------------------------------------------*/

   counter_buffer[index_CB].extended_tag <= extended_tag(cache_line);
   counter_buffer[index_CB].counter <= cache_line.counter;

   /*---------------------------------------------------------------*/
   /*   "event" can be one of {"V", "E", "R"}.                      */
   /*                                                               */
   /*        "V" = overflow of counter                              */
   /*        "E" = ejection of cache line due to eviction or        */
   /*              invalidation                                     */
   /*        "R" = (remotely) reading local cache line              */
   /*---------------------------------------------------------------*/

   counter_buffer[index_CB].SF <= event;

   index_CB++;
}

establish-checkpoint0
{
   CSB.CF <= TENTATIVE CHECKPOINT_IS_ACTIVE;
   wait until all pending memory accesses are completed or negatively
      acknowledged;
   negatively acknowledge all cache-coherence messages until checkpoint is
      established;
   establish-tentative-checkpointo;

   CSB.CF <= PERMANENT-CHECKPOINT-IS-ACTIVE;
   establish_permanent_checkpoint();

   /*---------------------------------------------------------------*/
   /*   Established checkpoint.                                     */
   /*---------------------------------------------------------------*/

   CSB.CF <= CHECKPOINT-IS_NOT_ACTIVE;
}

establish-tentative-checkpoint0
{
   /*---------------------------------------------------------------*/
   /*   CSB.toggle_flag toggles between 0 and 1.                    */
   /*---------------------------------------------------------------*/
   i <= 1 - CSB.toggle_flag;
   save tag, status_of_line, line_of_data, and IDF of all cache lines
      into CSB.checkpoint_area[i].cache;
   save internal state of processor
      into CSB.checkpoint_area[i].processor_state;
   CSB.checkpoint_area[i].status <= TENTATIVE-CHECKPOINT-AREA;

}

establishgermanent-checkpoint0
{
   /*--------------------------------------------------------------*/
   /*   Establish permanent checkpoint.                            */
   /*--------------------------------------------------------------*/

   index_LB_ <= 0;
   index_CB_ <= 0;

   index_LB <= 0;
   index_CB <= 0;

   for (each cache-line in cache) {
      cache_line.counter <= 0;
   }

   i <= CSB.toggle_flag;
   CSB.checkpoint_area[i].status <= NULL;

   i <= 1 - CSB.toggle_flag;
   CSB.checkpoint_area[i].status <= PERMANENT-CHECKPOINT-AREA;
   CSB.toggle_flag <= i;
}

complete-permanent-checkpoint0
{
   /*--------------------------------------------------------------*/
   /*   Establish permanent checkpoint.                            */
   /*--------------------------------------------------------------*/

   index_LB_ <= 0;
   index_CB_ <= 0;

   index_LB <= 0;
   index_CB <= 0;

   for (each cache-line in cache) {
      cache_line.counter <= 0;
   }

   j <= CSB.toggle_flag;
   if ((CSB.checkpoint_area[j].status == PERMANENT-CHECKPOINT-AREA) &&
       (CSB.checkpoint_area[1 - j].status == TENTATIVE-CHECKPOINT-AREA)) {

      i <= CSB.toggle_flag;
      CSB.checkpoint_area[i].status <= NULL;

      i <= 1 - CSB.toggle_flag;
      CSB.checkpoint_area[i].status <= PERMANENT-CHECKPOINT_AREA;
      CSB.toggle_flag <= i;
   }
   else if ((CSB.checkpoint_area[j].status == NULL) &&
       (CSB.checkpoint_area[1 - j].status == TENTATIVE-CHECKPOINT-AREA)) {

      i <= 1 - CSB.toggle_flag;
      CSB.checkpoint_area[i].status <= PERMANENT-CHECKPOINT-AREA;
      CSB.toggle_flag <= i;
   }
   else if ((CSB.checkpoint_area[j].status == NULL) &&
       (CSB.checkpoint_area[1 - j].status == PERMANENT-CHECKPOINT-AREA)) {

      CSB.toggle_flag <= 1 - j;
   }
   else {
      do nothing special;



fault detection
---------------
if (RLC detects fault in processor module) {

   if (fault == permanent) {
      replace processor module with spare processor module;
      reset spare processor module, invalidating all entries
         in both lst-level cache and 2nd-level cache;
   }
   else {
      reset processor module, invalidating all entries
         in both lst-level cache and 2nd-level cache;
   }

   trap to virtual machine monitor;

   query all memory modules to find lost cache-coherence messages;
   negatively acknowledge all cache-coherence messages
      until recovery is complete;

   if (CSB.CF == PERMANENT-CHECKPOINT-IS-ACTIVE) {
      complete_permanent_checkpoint();

      i <= CSB.toggle_flag;
      if (CSB.checkpoint_area[i].status != PERMANENT-CHECKPOINT-AREA) {
         i = 1 - CSB.toggle_flag;
      }

      load internal state of processor
         from CSB.checkpoint_area[i].processor_state;
      for (each cache line in cache) {
         load cache-line from CSB.checkpoint_area[i].cache;

         cache_line-counter <= 0;
      }

      return from trap to virtual machine monitor;
      exit and resume normal execution;
   }
   if (CSB.CF == TENTATIVE-CHECKPOINT-IS-ACTIVE) {
      i <= 1 - CSB.toggle_flag;
      CSB.checkpoint_area[i].status <= NULL;

      CSB.CF <= CHECKPOINT_IS NOT_ACTIVE;

      discard tentative checkpoint;
   }

   read all valid entries from line buffer;
   group all entries according to cache index but, for each cache index,
      maintain the temporal order in which the entries were originally
         inserted into the line buffer;
   place grouped entries into sorted-line-buffer;

   read all valid entries from counter buffer;
   group all entries according to cache index but, for each cache index,
     maintain the temporal order in which the entries were originally
         inserted into the counter buffer;
   place grouped entries into sorted-counter-buffer;

   i <= CSB.toggle_flag;
   if (CSB.checkpoint_area[i].status != PERMANENT-CHECKPOINT-AREA) {
      i = 1 - CSB.toggle_flag;
   }

   load internal state of processor
      from CSB.checkpoint_area[i].processor_state;
   for (each cache line in cache) {
      load cache-line from CSB.checkpoint_area[i].cache;

      cache line.counter <= 0;
      cache_line.SF <= "V";
   }

   return from trap to virtual machine monitor;

   enter recovery mode of execution;
}

execution mode:  recovery
-------------------------
switch (event) {

   /*------------------------------------------------------------------*/
   /*   Upgrade miss occurs in cache data line.                         */
   /*                                                                  */
   /*   Upgrade miss occurs when data-write "hits" in cache            */
   /*        line with status being SHARED.                            */
   /*------------------------------------------------------------------*/

   data-write-has-upgrade-miss-in-cache-data-line:  {
      stall  data-write:

      cache_line.status_of_line <= EXCLUSIVE;

      /*--------------------------------------------------------------*/
      /*   Retry data-write.                                          */
      /*                                                              */
      /*   It then will hit in cache data line and will generate      */
      /*        "data-access-hits_in-cache-data-line" as next         */
      /*        event.                                                */
      /*--------------------------------------------------------------*/

      retry  data-write;

      break;
   }

   /*-------------------------------------------------------------------*/
   /*   Access misses in cache data line.                               */
   /*-------------------------------------------------------------------*/

   data-access-misses-in-cache-data-line:  {
      stall data access;

      trap to virtual machine monitor;
      cache-line <= available_cache_line();
      get_entry_from_sorted_line_buffer(cache_line)
      get-entry-from-sorted_counter_buffer(cache_line);
      return from trap to virtual machine monitor;

      retry data access;

      exit_recovery_upon_completion();

      break;
   }

   memory-block-arrives:  {
      if (original access is data access) {

         /*-------------------------------------------------------------*/
         /*   Data is supplied from sorted line buffer.                 */
         /*-------------------------------------------------------------*/

         ;
      }
      else {
         cache-line <= available_cache_line();
         install_memory_block_into_cache_instruction_line();
      }
      break;
   }

   /*-------------------------------------------------------------------*/
   /*   Access hits in cache line.                                      */
   /*-------------------------------------------------------------------*/

   data-access-hits-in-cache-data-line:  {
      switch (cache_line.SF) {

         "N": {
            cache_line.counter <= 0;
            break;
      }

      "E": {
         if (cache-line.counter != 0) {
            cache_line-counter--;
         }
         else {
            stall data access;

            trap to virtual machine monitor;
            cache_line-status_of_line <= INVALID;
            get_entry_from_sorted_line_buffer(cache_line)
            get-entry-from-sorted counter_buffer(cache_line);
            return from trap to virtual machine monitor;

            retry data access;
         }
         break;
      }

      "R": {
         if (cache_line.counter != 0) {
            cache_line.counter--;
         }
         else {
            stall data access;

            trap to virtual machine monitor;
            cache line.status of line <= SHARED;
            get-entry-from-sorted counter buffer(cache line);
            return from trap to virtual machine monitor;

            retry data access;
         }
         break;
      }

      "V": {
         if (cache-line-counter != 0) {
            cache_line.counter--;
         }
         else {
            stall data access;

            trap to virtual machine monitor;
            get-entry-from-sorted_counter_buffer(cache_line);
            return from trap to virtual machine monitor;

            retry data access;
         }
         break;

   }
   exit_recovery_upon_completion();
   break;
}

default: {
```

```
            exit-recovery-upon-completion();
            break;
        }
    }

    write_back-and-invalidate-cache-lines0
    {
        i <= CSB.toggle_flag;
        if (CSB.checkpoint_area[i].status != PERMANENT-CHECKPOINT-AREA) {
            i <= 1 - CSB.toggle_flag;
        }

        for (each dirty cache-line in CSB.checkpoint_area[i].cache) {
            write cache-line back into main memory;
        }
        for (each cache-line in CSB.checkpoint_area[i].cache) {
            cache_line.status_of_line <= INVALID;
        }

        tell the directory controller of each memory module
            to change the status of any memory block (i. e. cache line)
            held by the cache (of the local processor) to indicate that
            the cache no longer holds the memory block;

        for (each cache-line in cache of local processor) {
            cache line.status_of_line <= INVALID;
        }
    }

    exit-recovery-upon-completion()
    {
        if ((sorted-counter-buffer has no entry where SF is "E" or "R") &&
           (counters in all valid cache data lines where SF is "E" or "R" are 0)) {

            /*-------------------------------------------------------------*/
            /*   Completion of recovery is imminent.                        */
            /*                                                              */
            /*   Update state of cache.                                     */
            /*-------------------------------------------------------------*/

            for (each valid cache-line in cache) {
                switch (cache-line.SF) {
                    "E": {
                        cache_line.status_of_line <= INVALID;
                        break;
                    }
                    "R": {
                        cache_line.status_of_line <= SHARED;
                        break;
                    }
                    default: {
                        break;
                    }
                }
            }

            establish_checkpoint();
        write-back-and-invalidate_cache_lines();

        /*---------------------------------------------------------------*/
        /*   Recovery is complete.                                       */
        /*---------------------------------------------------------------*/

        exit recovery and resume normal execution;

    }
    else {

        /*---------------------------------------------------------------*/
        /*   Completion of recovery is not imminent.                     */
        /*---------------------------------------------------------------*/

        ;
    }
}

get-entry-from-sorted_line_buffer(cache-line)
{
    /*-------------------------------------------------------------------*/
    /*   Get matching entry from sorted-line-buffer.                     */
    /*-------------------------------------------------------------------*/

    get next matching entry from sorted-line-buffer;

    cache-line.tag <= tag(sorted_line_buffer[entry].extended_tag);
    if (data-access == write) {
        cache_line.status_of_line <= EXCLUSIVE;
    }
    else
        cache_line.status_of_line <= SHARED;
    }
    cache-line-line-of-data <=sorted_line_buffer[entry].line_of_data;
}


get-entry-from-sorted-counter(cache-line)
{
    /*-------------------------------------------------------------------*/
    /*   Get matching entry from sorted-counter-buffer.                  */
    /*-------------------------------------------------------------------*/

    get next matching entry from sorted-counter-buffer;

    if (no matching counter) {
        cache line.counter <= 0;
        cache-line.IDF <= 1;
        cache_line.SF <= "N";
    }
    else {
        cache-line.counter  <=  sorted-counter-buffer[entry].counter;
        cache_line.IDF <= 1;
        cache_line.SF <= sorted-counter-buffer[entry].SF;
    }

}
```

H. Pedagogical Example

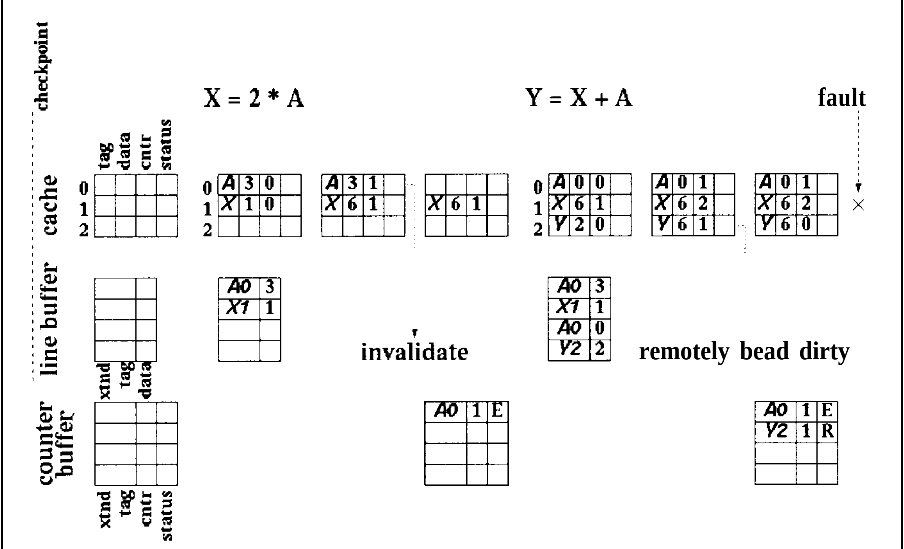


Figure 10. Normal Execution of Processor

To complete our description of DRSM-L, we illustrate its operation with a simple example. Figure 10 illustrates the normal execution of a processor "P" from the last checkpoint. At the last checkpoint, the 2nd-level cache, the line buffer, and the counter buffer are empty. The 2nd-level cache is a directly mapped cache with 3 entries. We designate the 2nd-level cache as simply "cache" in figure 10. As for the fields of the cache, we designate the "line of data", the "counter", and the 2-bit status flag (SF) as simply "data", "cntr". and "status". In the line buffer and the counter buffer, we designate the "extended tag" as "xtnd tag". For simplicity, we omit "status of line" and the instruction/data flag (IDF) from the cache. Also, we do not consider instruction accesses. We consider only data accesses.

In this example, "P" executes 2 simple statements: "X = 2 * A" and "Y = X + A". When "P" attempts to execute "X = 2 * A", the data accesses for "A" and "X" miss in the 2nd-level cache. After the incoming memory blocks satisfying these accesses arrive at the local directory controller, it forwards them to the 2nd-level cache and, avoiding any additional delay, concurrently copies them into the line buffer. The cache resets the counters of the affected cache lines to 0. Then, "P" reads the value of "A", calculates the new value of "X", and writes that value, 6, into the cache line. Since "P" accesses each of "A" and "X" once, the cache increments the counter of each of "A" and "X" by 1.

Then, a remote processor writes into the cache line for "A". The local directory controller receives an invalidation and forwards it to the 2nd-level cache. It invalidates the cache line for "A". The cache sends an acknowledgment to the directory controller and, avoiding any additional delay, concurrently sends the counter (of the invalidated cache line) with the acknowledgment. The directory controller inserts the counter into the counter buffer and sets the SF to "E", indicating that an eviction/invalidation of the cache line occurred after the number of accesses recorded by the counter.

Next, "P" executes "Y = X + A". When "P" attempts to execute "Y = X + A", the data accesses for "A" and "Y" miss in the 2nd-level cache. After the incoming memory blocks satisfying these accesses arrive at the local directory controller, it forwards them to the 2nd-level cache and, avoiding any additional delay, concurrently copies them into the line buffer. The cache resets the counters of the affected cache lines to 0. Then, "P" reads the values of "A" and "X", calculates the new value of "Y", and writes that value, 6, into the cache line. Since "P" accesses each of "A", "X", and "Y" once, the cache increments the counter of each of "A", "X", and "Y" by 1.

Then, a remote processor reads the memory block of "Y". The local directory controller receives a write-back request (associated with the remote read) and forwards it to the 2nd-level cache. It changes the status of the affected cache line from EXCLUSIVE to SHARED. The cache sends a copy of both the data and the counter in the line to the directory controller and concurrently resets the counter to 0. The directory controller inserts the counter into the counter buffer and sets SF to "R", indicating that a remote read of the cache line occurred after the number of accesses recorded by the counter.

Now, we suppose that a fault occurs at this point. Figure 11 illustrates how "P" performs recovery and, specifically, how "P" reproduces the exact same dirty value that "P" produced prior to the fault.

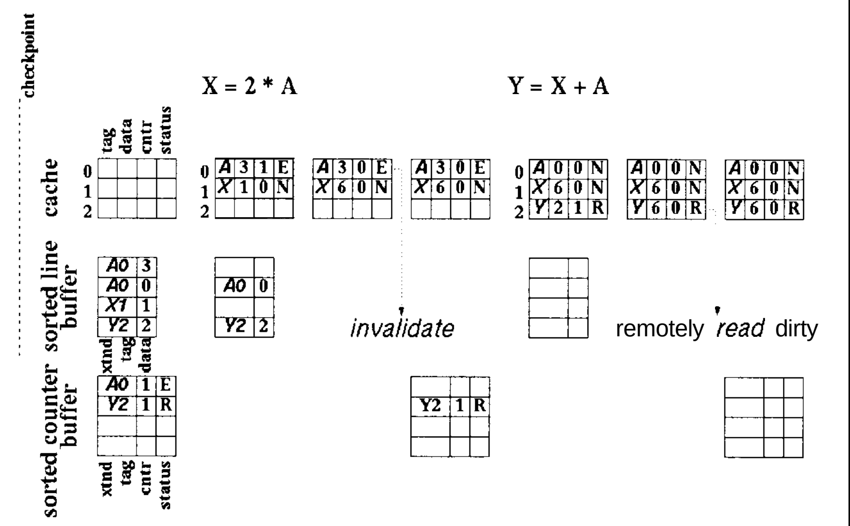


Figure 11. Recovery of Processor

The recovery logic circuit (RLC) resets "P". (If "P" experienced a permanent fault, the RLC replaces the failed processor module with a spare processor module, and the spare processor becomes "P".) "P" enters the virtual machine monitor (VMM). It loads the 2nd-level cache with the contents of the cache saved at the last checkpoint. For each line in the cache, the VMM resets the counter to 0 and sets the SF to "V". (In our simple example, the contents of the cache saved at the last checkpoint is empty and has no valid lines.) The VMM also loads "P" with the processor state saved at the last checkpoint. Next, the VMM groups the entries (by the 2nd-level-cache index of the extended tag) of the line buffer and the counter buffer and places the entries into the sorted line buffer (SLB) and the sorted counter buffer (SCB). The grouping procedure maintains, for each cache index, the temporal order in which the entries were originally inserted into the line buffer and the counter buffer.

Then, "P" proceeds to execute in recovery mode. When "P" attempts to execute "X = 2 * A", the data accesses for "A" and "X" miss in the 2nd-level cache. Execution traps to the VMM. To satisfy each of these misses, the VMM retrieves the next matching entry from the SLB and the next matching entry from the SCB and places the contents of the entries into the appropriate cache line. Since the SCB has no matching entry for "X", the VMM resets the counter in the cache line for "X" to 0 and sets the SF in that cache line to "N" (meaning "no event"). Then, "P" reads the value of "A", calculates the new value of "X", writes that value, 6, into the cache line. Since "P" accesses each of "A" and "X" once, the cache decrements the counter of each of "A" and "X" by 1 if counter is not 0. If the counter is already 0, the counter is not decremented.

At this point in normal execution, an invalidation arrives at the 2nd-level cache. Figure I l illustrates this event with an italicized label.

Next, "P" executes "Y = X + A". When "P" attempts to execute "Y = X + A", the data access for "A" hits in the 2nd-level cache. It discovers that the counter for "A" is 0, and since SF is not "N", execution traps to the VMM. It then acts on the value of the SF. Since it is "E" and indicates that the cache line was evicted/invalidated, the VMM retrieves the next matching entry from the SLB and the next matching entry from the SCB and places the contents of the entries into the cache line for "A". Since the SCB has no matching entry for "A", the VMM resets the counter in the cache line for "A" to 0 and sets the SF in that cache line to "N" (meaning "no event").

The data access for "X" also hits in the 2nd-level cache. It discovers that the counter for "X" is 0, but since the SF is "N", execution does not trap to the VMM.

The data access for "Y" misses in the cache. To handle the access miss on "Y", the VMM retrieves the next matching entry from the SLB and the next matching entry from the SCB and places the contents of the entries into the cache line for "Y".

Then, "P" reads the values of "A" and "X", calculates the new value of "Y", and writes that value, 6, into the cache line. Since "P" accesses each of "A", "X", and "Y" once, the cache decrements the counter of each of "A", "X", and "Y" by 1 <u>if the counter is not 0</u>. If the counter is already 0, the counter is not decremented.

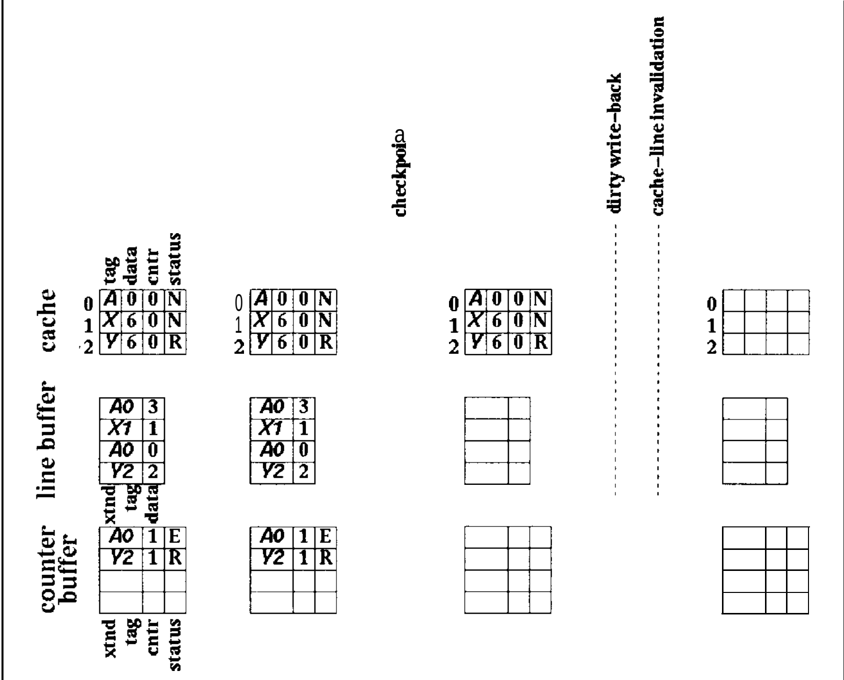


Figure 12. Completion of Recovery of Processor

At this point in normal execution, a write-back request (due to a read by a remote processor) arrives at the 2nd-level cache. Figure 11 illustrates this event with an italicized label. We note that "P" has reproduced the exact same dirty value that "P" produced prior to the fault.

Currently, "P" has reached the state of "imminent completion of recovery". "P" has achieved the following 2 conditions: (I) the SCB has no remaining entry with a SF of "E" or "R" and (2) the counters in all valid cache data lines (i. e. with IDF being "1") with a SF of "E" or "R" are 0. Execution traps to the VMM. For each valid cache data line, if the SF is "E" or "R". then the VMM changes the status of the line to INVALID or SHARED, respectively.

To complete the recovery from the fault, the VMM invokes "P" to establish a checkpoint, indicated in figure 12. Establishing the checkpoint clears the line buffer and the counter buffer and resets all the counters in the 2nd-level cache to 0. Once the checkpoint is established, the VMM reads each dirty 2nd-level-cache line saved in the permanent-checkpoint area of the CSB and writes the line back into main memory. The VMM changes the status of each 2nd-level-cache line saved in the permanent-checkpoint area (of the CSB) to INVALID. Then, the VMM tells the directory controller of each memory module to change the status of any memory block (i. e. cache line) held by the 2nd-level cache (of "P") to indicate that the 2nd-level cache no longer holds the memory block. The VMM also changes the status of each line in the 2nd-level-cache (of "P") to INVALID. The contents of the 2nd-level cache are now consistent with the information stored in the directories of the memory modules. “P” resumes normal execution.

“P” must establish a checkpoint at the end of recovery in order to deal with the loss of counter values after a fault. The last values of the counters for “A” and “X” in figure 10 are 1 and 2, respectively. Both 1 and 2 are lost after the fault occurs.

V. Optimal Size of Line Buffer and Counter Buffer

For a given amount of silicon area from which we can build the line buffer and counter buffer, we show that the optimal size of each is one where the ratio of the number of entries in the counter buffer to the number of entries in the line buffer equals the ratio of the rate at which the counter buffer fills to the rate at which the line buffer fills. Suppose that we have the following parameters.

```
   E[CB]  = number of entries in the counter buffer
   E[LB] = number of entries in the line buffer
A(T, E) = amount of silicon area consumed by transistors to implement "E"
           entries for buffer of type “T”
      AA = fixed amount of allocated silicon area in which to implement
           counter buffer and line buffer
   R[CB] = rate at which counter buffer fills in terms of the number of
           entries per unit time
   R[LB] = rate at which line buffer fills in terms of the number of entries
           per unit time
      RC = rate of establishing checkpoints
```

Then, considering only checkpoints triggered by overflowing a buffer, we have the following equations.

```
A(“counter buffer", E[CB]) + A(“line buffer", E[LB]) = AA        (equation #1)

          RC = max (R[CB] / E[CB], R[LB] / E[LB])                (equation #2)
```

The optimum size of each of the counter buffer and the line buffer arises when the "RC", rate of establishing checkpoints, is minimum. Suppose that we select "E[CB]" and "E[LB]" to be "E0[CB]" and "E0[LB]", respectively, where

$$RC = \max(R[CB] / E0[CB], R[LB] / E0[LB]) \quad \text{(equation \#3)}$$

$$= R[CB] / E0[CB] = R[LB] / E0[LB]. \quad \text{(equation \#4)}$$

Now, we consider what happens when we increase "E[CB]" or decrease it. Suppose that we increase it to some value "E2[CB]" such that "E2[CB]" is greater than "E0[CB]". By equation #1, "E[LB]" must decrease to some value, say "E2[LB]. Then, we have that

$$RC = \max(R[CB] / E2[CB], R[LB] / E2[LB]) \quad \text{(equation \#5)}$$

$$= R[LB] / E2[LB]. \quad \text{(equation \#6)}$$

On the other hand, suppose that we decrease "E[CB]" to some value "E1[CB]"such that "E1[CB]" is less than "E0[CB]". By equation #1, "E[LB]" must increase to some value, say "E1[LB]. Then, we have that

$$RC = \max(R[CB] / E1[CB], R[LB] / E1[LB]) \quad \text{(equation \#7)}$$

$$= R[CB] / E1[CB]. \quad \text{(equation \#8)}$$

Comparing equation #4, equation #6, and equation #8, we see that "RC" is smallest when "E[CB]" equals "E0[CB]". Hence, the optimum ratio of "E[CB]" to "E[LB]" is one where

$$E[CB] / E[LB] = R[CB] / R[LB]. \quad \text{(equation \#9)}$$

VI. Simulation Environment and Methodology

A. Multiprocessor Simulator

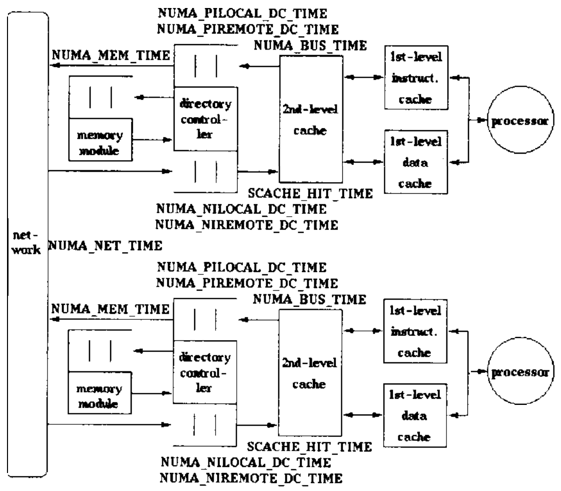


Figure 13. Base Multiprocessor

We evaluated DRSM-L by simulating its operation within a multiprocessor simulator. The block diagram of the base multiprocessor in our simulator appears in figure 13, illustrating a 2-processor configuration. The model of the memory system and the network is the NUMA model packaged with the SimOS simulator [4]. Instead of SimOS, we use our own simulator, ABSS, to simulate the processors and to drive the model of the memory system and the network. ABSS is an augmentation-based simulator that runs significantly faster than SimOS [8]. Our simulator has the following parameters.

```
base parameters
processor = SPARC V7 @ 200 megahertz
cache policy = write-back
memory model = sequential consistency

lst-level instruction cache = 32 kilobytes with 4-way set associativity,
                             2 states (INVALID, SHARED), 64-byte line
Ist-level data cache = 32 kilobytes with 4-way set associativity,
                      3 states (INVALID, SHARED, EXCLUSIVE), 64-byte line
2nd-level cache = 1 megabyte with 4-way set associativity,
                 3 states (INVALID, SHARED, EXCLUSIVE), 128-byte line

average delay (NUMA BUS-TIME) between 2nd-level cache
              and directory controller (DC) = 75 cycles
average delay (SCACHE_HIT_TIME) for access that hits
              in the 2nd-level cache = 50 cycles
average delay (NUMA_PILOCAL_DC_TIME) in the local DC
              for local access = 100 cycles
average delay (NUMA PIREMOTE-DC-TIME) in the local DC
              for remote access = 25 cycles
average delay (NUMA_NILOCAL_DC_TIME) in the remote DC
              for remote access = 350 cycles
average delay (NUMA_NIREMOTE_DC_TIME) in the remote DC
              for remote reply = 25 cycles
average network delay (NUMA_NET_TIME) between 2 DCs = 150 cycles
average delay (NUMA_MEM TIME) to access memory = 50 cycles

DRSM-L parameters
width of counter = 32 bits
line buffer = 8192 entries
counter buffer = 8192 entries
timer = expiration per 20 million cycles
```

B. Benchmarks

In ABSS, We run 6 benchmarks -- Cholesky, FFT, LU, ocean, radix, and water -- from the SPLASH2 suite [12] Cholesky factors a sparse matrix. FFT performs a fast Fourier transform. LU factors a dense matrix. Ocean simulates eddy and boundary currents in oceans. Radix performs a radix sort. Finally, water evaluates the forces and potentials as they change overtime among water molecules.

Woo presents a detailed study of these benchmarks [12]. They have 2 common characteristics. First, the working set of all these benchmarks fit within the large 2nd-level cache of our TCMP. Second, these benchmarks represent a scientific workload. They are useful in representing a wide variety of memory-access patterns but do virtually no communication with the environment outside of the TCMP. So, establishing a checkpoint that is triggered by communication between a processor and the environment outside of the TCMP does not arise in our simulations. We note that regardless of the event triggering the establishment of a checkpoint, the procedure for establishing a checkpoint remains the same. Hence. we can still evaluate the performance of our hardware-based algorithms even if checkpoint establishment is tri ggered by a smaller set of events.

C. LSM-type Algorithm and Apparatus

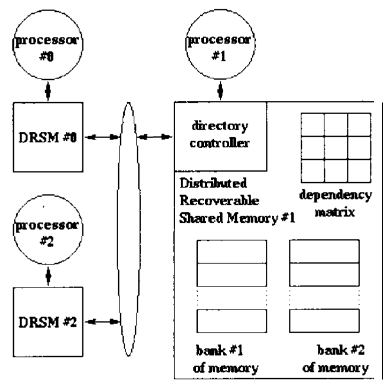


Figure 14. Distributed Recoverable Shared Memory

In order to compare the performance of DRSM-L (as a USM-type algorithm) against a LSM-type algorithm, we introduce DRSM. It is a recently developed LSM-type algorithm and is an extension of recoverable shared memory (RSM) developed by Banatre. Figure 14 illustrates only the key structures of DRSM for a 3-processor configuration.

We very briefly describe how DRSM works. Bank #1 of memory holds the working data, and bank #2 holds the permanent checkpoint. The dependency matrix records checkpoint dependencies that can arise when 2 processors access the same memory block in the memory module. For example, if processor P1 reads a memory block into which P2 previously wrote data, then the checkpoint dependency "P1 -> P2" arises, and the directory controller sets entry [1, 2] of the matrix to " 1".

DRSM establishes a checkpoint in 2 phases: tentative checkpoint and permanent checkpoint. Suppose that processor "P" wishes to establish a tentative checkpoint. "P" establishes it, then queries all the dependency matrices, and identifies all processors that are dependent on "P". "P" tells them to join the tentative checkpoint. As a recursive step, each dependent processor, in turn, queries the dependency matrices, determines all dependent processors, and tells them to join the tentative checkpoint. DRSM repeats the recursive step until it finds all dependent processors. A processor joins the tentative checkpoint by saving the processor state into the local memory module and by writing all dirty 2nd-level-cache lines back into main memory (i. e. bank #1).

Once the tentative checkpoint is concluded, "P" initiates the permanent checkpoint by telling all dependent processors and all dependent memory modules to convert the tentative checkpoint into a permanent checkpoint. A dependent memory module is a memory module where (1) the dependency matrix has non-zero entries for any dependent processor or "P" or (2) bank #1 has dirty blocks written by any dependent processor or "P" since the last checkpoint. Each dependent memory module marks each block containing tentative-checkpoint data in bank #I to indicate that the tentative-checkpoint data is converted into permanent-checkpoint data. After the permanent checkpoint is concluded, a write access hitting on a marked block causes the memory module to first copy the data (in the marked block) into bank #2 before answering the write access. After the dependent processors and the dependent memory modules complete their permanent checkpoint, “P” completes its permanent checkpoint.

The DRSM described here differs from the DRSM in prior work [10] in regards to only 1 aspect. In the current DRSM, bank #2 always holds the permanent checkpoint, but in the prior DRSM, bank #2 alternates between holding permanent-checkpoint data and holding working data. The extra functionality in the prior DRSM proved unnecessary, so we removed the functionality and simplified the hardware.

Finally, only 2 events trigger the establishment of a checkpoint. They are (1) expiration of a timer and (2) communication between a processor and the environment outside of the TCMP.

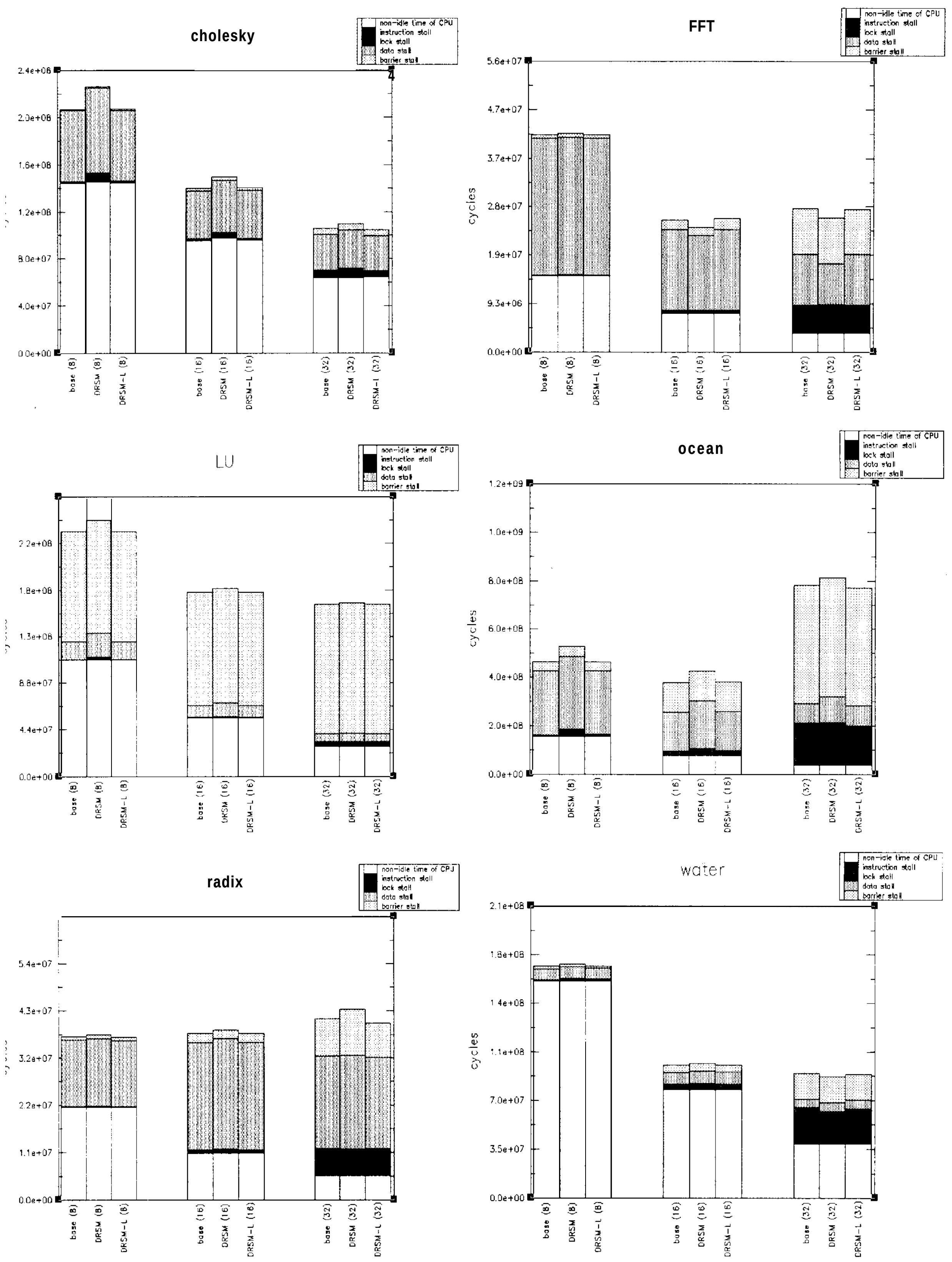


Figure 15. Performance of 6 Benchmarks

VII. Analysis and Results

A. Overall Performance of Benchmarks

Figure 15 shows the performance of the 6 application benchmarks running on 3 TCMPs: the base system, the system with DRSM, and the system with DRSM-L. We set the number of processors to 8, 16, and 32. We decompose the execution time into 5 categories: non-idle time of the processor, the instruction stall, the lock stall, the data stall, and the barrier stall. In general, the performance of DRSM-L exceeds the performance of DRSM.

We see 2 notable effects. First, for all benchmarks except Cholesky, the barrier stall and the lock stall increase substantially as the number of processors increases from 8 to 32 processors because all benchmarks except Cholesky have several global barriers and global locks. Both the locks within global barriers and the global locks, where all processors compete for a lock, cause hot spots to arise at the memory addresses holding the locks. In Cholesky, after the processors enter the main loop of execution, they encounter no global synchronization. Hence, Cholesky does not suffer this problem.

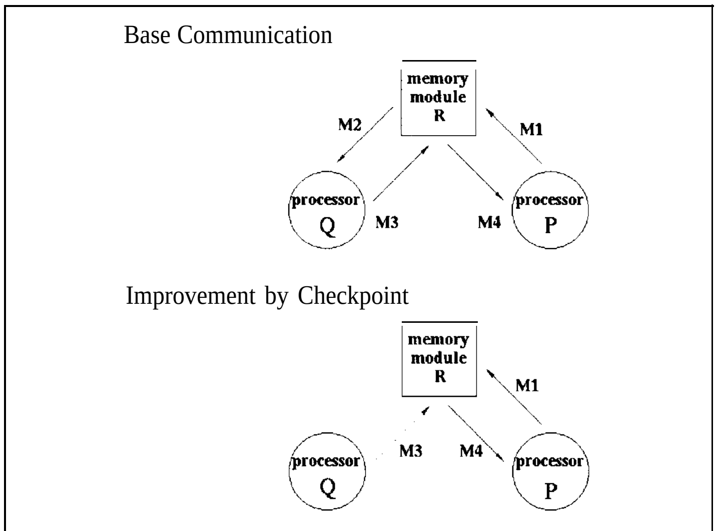


Figure 16. Communication Improved by Checkpoint

Second, the checkpointing algorithm can occasionally cause a TCMP with DRSM to exceed the performance of the base TCMP. Figure 16 illustrates the explanation for this effect. In the base system, the transfer of dirty data from processor "Q" to processor "P" typically involves the following activities. "P" suffers a read miss in the local cache. "P" sends message "M1" to the remote memory module "R". It sends a request "M2" to "Q" to retrieve the dirty data. "Q" replies to "R" with message "M3". "R" forwards the data to "P" in message "M4". This type of communication involves 4 messages: "M 1 ", "M2", "M3", and "M4".

Now, consider a TCMP with DRSM. Suppose that "Q" establishes a checkpoint just prior to this communication. the transfer of dirty data. After the checkpoint, "P" reads the dirty data that was in the cache of

"Q". "P" suffers only 2 messages: “MI " and "M4". The net cost of the communication itself is 3 messages: “Ml ", "M3", and "M4". ("M3" is the cost that is part of "Q" establishing a checkpoint.) Therefore, the checkpointing improved the performance of the communication by eliminating the cost of message "M2". In order for this benefit to have maximum impact, the checkpointing must occur just before such transfer of dirty data, but DRSM does not guarantee that checkpointing will occur at such an opportune time. Hence, in figure 15, TCMP with DRSM only occasionally -- not always -- performs better than the base TCMP.

B. Checkpoints and Checkpoint Data

	8-processor TCMP	16-processor TCMP	32-processor TCMP	
Cholesky	10.38	7.00	5.19	checkpoints
	(9.62 + 0.12 + 0.62)	(6.94 + 0.00 + 0.06)	(5.03 + 0.12 + 0.03)	(please see text)
	(8.189 + 0.106 + 0.532)	5.902 + 0.000 + 0.055)	4.281 + 0.106 + 0.027)	x 1e+4 cycles
	(3.977 + 0.052 + 0.258)	4.229 + 0.000 + 0.040)	4.109 + 0.102 + 0.026)	x 0.01 % of run time
FFT	2.00	1.31	1.34	checkpoints
	(1.88 + 0.00 + 0.12)	(1.25 + 0.00 + 0.06)	(1.31 + 0.00 + 0.03)	(please see text
	(1.595 + 0.000 + 0.106)	1.063 + 0.000 + 0.058)	1.117 + 0.000 + 0.030)	x 1e+4 cycles
	(3.831 + 0.000 + 0.255)	(4.146 + 0.000 + 0.224)	4.060 + 0.000 + 0.109)	x 0.01 % of run time
LU	7.50	4.88	4.12	checkpoints
	(7.50 + 0.00 + 0.00)	(4.81 + 0.00 + 0.06)	(4.09 + 0.00 + 0.03)	(please see text)
	(6.381 + 0.000 + 0.000)	(4.094 + 0.000 + 0.053)	(3.483 + 0.000 + 0.047)	x 1e+4 cycles
	(2.752 + 0.000 + 0.000)	(2.341 + 0.000 + 0.031)	(2.137 + 0.000 + 0.029)	x 0.01 % of run time
ocean	22.38	18.19	37.25	checkpoints
	(21.38 + 0.25 + 0.75)	(18.19 + 0.00 + 0.00)	(37.25 + 0.00 + 0.00)	(please see text)
	(18.186 + 0.213 + 0.682)	(15.474 + 0.000 + 0.000)	(31.692 + 0.000 + 0.000)	x 1e+4 cycles
	(3.903 + 0.046 + 0.146)	(4.045 + 0.000 + 0.000)	(4.098 + 0.000 + 0.000)	x 0.01 % of run time
radix	1.00	1.00	1.03	checkpoints
	(0.88 + 0.12 + 0.00)	(0.38 + 0.62 + 0.00)	(0.62 + 0.41 + 0.00)	(please see text)
	(0.744 + 0.106 + 0.000)	(0.319 + 0.532 + 0.000)	(0.532 + 0.346 + 0.000)	x 1e+4 cycles
	(2.006 + 0.287 + 0.000)	(0.841 + 1.401 + 0.000)	(1.321 + 0.859 + 0.000)	x 0.01 % of run time
water	7.50	4.00	3.56	checkpoints
	(7.50 + 0.00 + 0.00)	(4.00 + 0.00 + 0.00)	(3.56 + 0.00 + 0.00)	(please see text)
	(6.381 + 0.000 + 0.000)	(3.403 + 0.000 + 0.000)	3.031 + 0.000 + 0.000)	x 1e+4 cycles
	(3.816 + 0.000 + 0.000)	(3.547 + 0.000 + 0.000)	3.415 + 0.000 + 0.000)	x 0.01 % of run time

Table 1. Checkpoints for DRSM-L

Table 1 shows statistics about the rate at which DRSM-L establishes checkpoints per processor for each of the 6 benchmarks. For each application, there are 4 rows of statistics. The first row indicates the total number of checkpoints established per processor. DRSM-L has effectively 3 events that trigger the establishment of a checkpoint; they are (1) timer expiration, (2) line-buffer overflow, and (3) counter-buffer overflow. The checkpoints that are attributed to each trigger appear in the 2nd row of statistics. For example, in the 2nd row for Cholesky running on an &processor TCMP, we see "(9.62 + 0.12 + 0.62)". The number of checkpoints due to timer expiration, line-buffer overflow, and counter-buffer overflow are 9.62, 0.12, and 0.62, respectively.

The remaining 2 rows show the time consumed by checkpoint establishment. The 3rd row shows the number of cycles for which a processor is stalled in establishing the number of checkpoints in the 2nd row. Each number within parentheses in the 3rd row indicates a fraction of 10,000 cycles. The 4th row shows the percentage (of the total execution time of the benchmark) represented by the number of cycles in the 3rd row. For example, during the execution of the Cholesky benchmark by the 8-processor TCMP, a typical processor consumed 81,890 cycles in establishing a total of 9.62 timer-triggered checkpoints. The 81,890 cycles is 0.03977 % of the total number of cycles needed to execute Cholesky.

The data for DRSM-L indicates that the 8 192-entry line buffer and the 8 192-entry counter buffer are adequately large. They overflow infrequently and, hence, trigger the establishment of checkpoints only infrequently. Based on the number of bits of storage, the size of the combination of the line buffer and the counter buffer is close to the size of the 2nd-level cache.

	E-processor TCMP	16-processor TCMP	32-processor TCMP	
Cholesky	11.00	8.00	6.00	checkpoints
	(11.00)	(8.00)	(6.00)	(please see text)
	(7.530)	(3.447)	(2.039)	x 1e+6 cycles
	(3.353)	(2.327)	(1.868)	% of run time
FFT	2.00	2.00	2.00	checkpoints
	(2.00)	(2.00)	(2.00)	(please see text)
	(0.700)	(0.278)	(0.193)	x 1e+6 cycles
	(1.667)	(1.161)	(0.751)	% of run time
LU	9.38	6.19	5.09	checkpoints
	(9.38)	(6.19)	(5.09)	(please see text)
	(2.775)	(0.832)	(0.280)	x 1e+6 cycles
	(1.143)	(0.466)	(0.171)	% of run time
ocean	24.00	20.00	39.00	checkpoints
	(24.00)	(20.00)	(39.00)	(please see text)
	(31.227)	(17.650)	(10.910)	x 1e+6 cycles
	(5.871)	(4.117)	(1.338)	% of run time
radix	1.00	1.00	2.00	checkpoints
	(1.00)	(1.00)	(2.00)	(please see text)
	(0.306)	(0.378)	(0.359)	x 1e+6 cycles
	(0.813)	(0.974)	(0.828)	% of run time
water	8.00	5.00	4.00	checkpoints
	(8.00)	(5.00)	(4.00)	(please see text)
	(0.471)	(0.238)	(0.179)	x 1e+6 cycles
	(0.280)	(0.247)	(0.204)	% of run time

Table 2. Checkpoints for DRSM

Table 2 shows statistics about the rate at which DRSM establishes checkpoints per processor for each of the 6 benchmarks. The checkpoints in table 2 are tri ggered by the expiration of the timer.

Table 3 shows statistics about the amount of data written into the line buffer and the counter buffer. Each row has 3 consecutive numbers enclosed within parentheses. The 1st number is the number of entries written into the line buffer. The 2nd number is the number of entries written into the counter buffer. The 3rd number is the ratio of the 2nd number to the 1st number. This ratio is the optimum ratio of the number of entries in the counter buffer to the number of entries in the line buffer, according to equation #9.

	8-processor TCMP	16-processor TCMP	32-processor TCMP
Cholesky	(35559.5; 28548.5; 0.80)	(24420.2; 16946.8; 0.69)	(18822.8; 11469.8; 0.61)
FFT	(8782.0; 6335.1; 0.72)	(4602.8; 3155.5; 0.69)	(2680.0; 1882.9; 0.70)
LU	(10571.5; 4475.9; 0.42)	(6794.8; 2813.2; 0.41)	(5293.3; 2147.0; 0.41)
ocean	(124516.0; 116525.0; 0.94)	(61962.0; 54849.1; 0.89)	(39862.0; 35659.4; 0.89)
radix	(8389.2; 5223.5; 0.62)	(12267.9; 10375.3; 0.85)	(10379.2; 8968.8; 0.86)
water	(4893.5; 3976.4; 0.81)	(5123.9; 4326.2; 0.84)	(4508.6; 3874.1; 0.86)

Table 3. Audit-Trail Data (entries in line buffer; entries in counter buffer; ratio)

For DRSM-L with 8, 16, and 32 processors, the ratios represented by the 3rd numbers are concentrated in the range of [0.62, 0.94], [0.69, 0.89], and [0.61, 0.89], respectively, excluding 3 atypical extreme values (i.e. 0.42, 0.41, and 0.41). That the ratios are concentrated in a somewhat narrow band over several rather different applications is opportune. We can then select and use the average ratio (according to a geometric average) to determine the relative sizes of the line buffer and the counter buffer, and this average ratio shall yield good system performance across all the benchmarks. The geometric averages of the ratios within the bands of [0.62, 0.94], [0.69, 0.89], and [0.61, 0.89] are 0.79, 0.80, and 0.79, respectively. Our selected ratio of 1.O -- ratio of 8192 entries in the counter buffer to 8192 entries in the line buffer -- is somewhat larger than these 3 geometric averages. (For our 6 benchmarks, the ratios change little as we vary the number of processors).

Ideally, we use the following algorithm to determine the optimum ratio of the counter-buffer size to the line-buffer size.

1. Let R = 1. "1" is our initial guess of the optimum ratio.
2. Set the ratio of the number of entries in the counter buffer to the number of entries in the line buffer to "R". (We must set the number of entries according to the amount of silicon area that we can allocate for building the buffers.)
3. Run a representative set of application programs for a TCMP with the number of processors that we intend to use.
4. For each run of each application, determine the ratio of the number of entries written into the counter buffer to the number of entries written into the line buffer.
5. From the list of ratios determined in step #4, create a pruned list by eliminating the atypical extreme ratios.
6. Using the values in the pruned list, determine the geometric average, "Q".
7. If "Q" is adequately close in value to "R", then we can use "R" as the optimum ratio of the counter-buffer size to the line-buffer size. Otherwise, if "Q" is not adequately close in value to "R", we set "R" to the value of "Q" and repeat the whole procedure starting from step #2.

C. Performance Impact of Establishing Checkpoints

When a processor establishes checkpoints, 2 types of interference can degrade the performance of the processor in both DRSM and DRSM-L. First, the processor must waste time in actually establishing the checkpoint. Second, establishing a checkpoint causes certain resources to be unavailable; a processor attempting to access such a resource receives a negative acknowledgment. For example, when some processor "P" establishes a checkpoint, "P" negatively acknowledges cache-coherence messages (like invalidations) indirectly sent from other processors.

A processor in DRSM also suffers a third type of interference. During the establishment of a checkpoint, "P" converts much dirty data (in state EXCLUSIVE) in the 2nd-level cache into clean data (in state SHARED) by writing it back into main memory. After "P" resumes execution after establishing the checkpoint, "P" wastes time in submitting many upgrade requests to memory in order to convert clean data (which was dirty prior to the checkpoint) back into dirty data so that "P" can resume writing into that data.

We note that identifying the precise portion (of each bar in figure 15) contributed by each of the types of interference is difficult. This identification is complicated by several issues. First, establishing a checkpoint can actually but unpredictably improve the performance of a processor, depending on when the checkpoint is established. (Consider figure 16.) In addition, the delay caused by each type of interference can be amplified by the data dependencies among processors. For example, suppose that there are 3 processors: "P", "Q", and "R". Suppose that "P" must wait on a result produced by "Q" and that "Q" must wait on a result produced by "R". "R" is the head processor of this chain: "R => Q => P". Just prior to producing the result needed by "Q", "R" establishes a checkpoint. The delay experienced by "R" in establishing the checkpoint is then propagated down the chain of processors to "P". All processors in this chain then experience the delay. In general, determining (1) the occurrence of such a chain, (2) its head processor, and (3) its members is extremely difficult. Hence, instead of identifying the precise portion (of each bar in figure 15) contributed by each type of interference, we focus on the overall performance of the checkpointing algorithm and on selected statistics.

Table 4 shows the number of negative acknowledgments (NAKs) and upgrade misses generated by the base TCMP and the TCMP with DRSM-L. For each of the benchmarks, the 1 st row shows the number of NAKs; the impact of the 2nd type of interference is the increase in NAKs over that of the base TCMP. The 2nd row shows the number of upgrade misses. A processor in DRSM-L does not suffer the 3rd type of interference.

	8 processors		16 processors		32 processors		
	base	DRSM-L	base	DRSM-L	base	DRSM-L	
Cholesky	127.2	148.1	331.2	340.2	736.4	665.9	negative ack.'s
	9036.1	9078.9	4954.6	4938.4	3268.2	3285.2	upgrade misses
FFT	86.6	95.2	278.9	269.0	911.0	908.4	negative ack.'s
	6374.4	6377.2	3427.0	3428.5	1754.6	1755.4	upgrade misses
LU	982.4	993.4	1848.3	1915.9	3239.0	3225.6	negative ack.'s
	2058.8	2058.5	1041.1	1042.2	519.2	520.6	upgrade misses
ocean	6222.2	6347.1	15936.6	16480.3	50931.9	50128.0	negative ack.'s
	41021.2	40980.4	28386.2	28430.9	14449.4	14511.8	upgrade misses
radix	66.9	67.9	225.5	240.1	969.9	926.4	negative ack.'s
	105.4	107.0	203.7	209.3	176.1	178.1	upgrade misses
water	399.6	440.0	954.2	1007.7	3042.1	2979.5	negative ack.'s
	884.6	888.5	1220.0	1226.8	704.3	705.8	upgrade misses

Table 4. Negative Acknowledgments and Upgrade Misses for DRSM-L

Table 5 shows the number of negative acknowledgments (NAKs) and upgrade misses generated by the base TCMP and the TCMP with DRSM. For each of the benchmarks, the 1 st row shows the number of NAKs; the impact of the 2nd type of interference is the increase in NAKs over that of the base TCMP. The 2nd row shows the number of upgrade misses; the impact of the 3rd type of interference is the increase in upgrade misses over that of the base TCMP. This large increase in the number of upgrade misses is one of the major reasons that DRSM performs worse than DRSM-L.

	8 processors		16 processors		32 processors		
	base	DRSM	base	DRSM	base	DRSM	
Cholesky	127.2	148.8	331.2	334.9	736.4	768.7	negative ack.'s
	9036.1	22662.6	4954.6	9533.7	3268.2	5447.2	upgrade misses
FFT	86.6	80.9	278.9	270.8	911.0	914.8	negative ack.'s
	6374.4	7534.2	3427.0	3509.8	1754.6	1779.7	upgrade misses
LU	982.4	748.6	1848.3	1807.8	3239.0	3214.4	negative ack.'s
	2058.8	9602.9	1041.1	2925.9	519.2	1093.8	upgrade misses
ocean	6222.2	6404.5	15936.6	16624.4	50931.9	51009.8	negative ack.'s
	41021.2	75475.0	28386.2	61812.2	14449.4	38453.7	upgrade misses
radix	66.9	64.5	225.5	239.6	969.9	1069.6	negative ack.'s
	105.4	231.9	203.7	559.4	176.1	319.3	upgrade misses
water	399.6	430.1	954.2	1025.7	3042.1	2844.7	negative ack.'s
	884.6	1964.0	1220.0	1707.4	704.3	962.2	upgrade misses

Table 5. Negative Acknowledgments and Upgrade Misses for DRSM

D. DRSM Versus DRSM-L

DRSM-L, a USM-type algorithm, has an inherent advantage over DRSM, a LSM-type algorithm. DRSM-L enables a processor "P" to establish a checkpoint without regard to any other processor. By contrast, in a system with DRSM, if "P" establishes a checkpoint, then all processors that are checkpoint dependent on "P" must also establish a checkpoint. Suppose that "P" tends to establish checkpoints at a much higher rate than the other processors. If the TCMP uses DRSM, then checkpoint dependencies between “P” and the other processors tend to cause the other processors to establish checkpoints at a high rate, degrading the performance of the TCMP. On the other hand, if the TCMP uses DRSM-L, the high rate of checkpoints by "P" does not cause the other processors to establish checkpoints at a high rate. Hence, DRSM-L has an inherent performance advantage over DRSM.

To quantitatively demonstrate this performance advantage, we artificially increase the rate at which processor #3 in our TCMP establishes checkpoints. We set the timer of processor #3 to expire after each interval of 2 million cycles, but we keep the current timer interval of 20 million cycles for the other processors. In other words, we increase, by a factor of 10, the rate at which processor #3 tends to establish timer-triggered checkpoints.

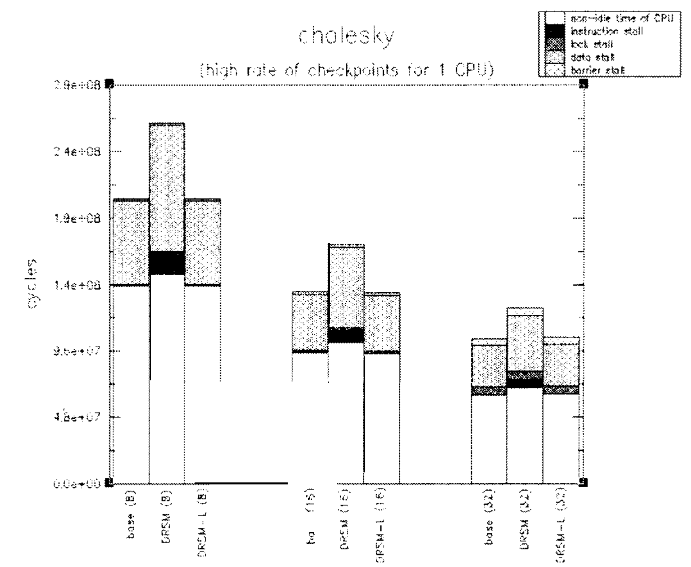


Figure 17. Effect of Irregular Checkpointing -- Processor #3 with High Checkpointing Rate

We focus on Cholesky because it, unlike the other benchmarks, does not suffer any hot spots. Figure 17 shows the overall results for Cholesky. (For DRSM, expiration of a timer is effectively the only event that triggers establishing a checkpoint.) In figure 15, DRSM-L runs about 9.09%, 6.13%, or 4.77% faster than DRSM for a TCMP with 8, 16, or 32 processors, respectively. In figure 17, DRSM-L runs about 26.75%, 25.00%, or 19.28% faster than DRSM for a TCMP with 8, 16, or 32 processors, respectively. DRSM performs much worse then DRSM-L in figure 17 because the high rate of establishing checkpoints by processor #3 causes the other processors to establish checkpoints at a high rate as well. To obtain insight into the extent to which checkpoint dependencies cause a high rate of checkpointing by one processor to impact other processors, we introduce a lumped parameter that is the average number of timer-triggered checkpoints across all processors <u>except</u> processor #3. Table 6 shows the values for this new parameter. Each row has 2 sets of numbers. The 1 st number in each set is the number of timer-triggered checkpoints established by processor #3. The 2nd number in each set is the average number of timer-triggered checkpoints across all processors <u>except</u> processor #3. Clearly, due to the checkpoint dependencies that are in DRSM, the high rate of establishing checkpoints by processor #3 causes all the other processors to establish checkpoints at almost the same high rate. Hence, DRSM performs much worse than DRSM-L.

	DRSM	DRSM-L	
8 CPUs	(120; 92.9)	(103; 9.6)	checkpoints
16 CPUs	(80; 68.8)	(68; 6.9)	checkpoints
32 CPUs	(58; 50.4)	(51; 5.1)	checkpoints

Table 6. Timer-triggered Checkpoints: (number for processor #3; average for other processors)

VIII. Conclusion

Compared to a base TCMP or a TCMP with DRSM, a TCMP with DRSM-L performs well. Specifically, we showed that DRSM-L performs significantly better than DRSM when 1 processor, processor #3 in our case, tends to establish checkpoints at a high rate. Such a scenario can arise when applications are structured in the following way. One processor, "P", in the TCMP performs I/O (i. e. communication) with the environment outside of the TCMP. Each of the other processors in the TCMP performs the core computation of the application and periodically sends results to "P". "P" sends many messages (containing those results) to the environment outside of the TCMP and, hence, must establish checkpoints frequently. When "P" does so, it will not interfere with the core computation being performed by the other processors -- if the TCMP uses DRSM-L.

We conclude that DRSM-L is a good checkpointing apparatus and algorithm for TCMPs. DRSM-L is the first USM-type algorithm for a TCMP. Unlike current algorithms, DRSM-L allows independent establishment of a checkpoint and independent roll-back from a fault and, hence, is much more scalable than DRSM. DRSM-L performs much better than DRSM. Also, DRSM-L is substantially cheaper to implement than DRSM. For example, DRSM-L requires only a single bank of memory, but both DRSM and RSM [2] require 2 banks of memory.

References

1. R. E. Ahmed, R. C. Frazier, et. al., "Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems", Proceedings of the 20th International Svmuosium on Fault-Tolerant Computing Systems, pp. 82-88, 1990.
2. M. Banatre, A. Gefflaut, et. al., "An Architecture for Tolerating Processor Failures in Shared-Memory Multiprocessors", "IEEE Transactions on Computers", vol. 45, no. 10, pp. 1101-l 11.5, October 1996.
3. E. Bugnion, S. Devine, et. al., "Disco: running commodity operating systems on scalable multiprocessors", "ACM Transactions on Computer Systems", vol. 15, no. 4, Pages 412-447, November 1997.
4. S. Herrod, M. Rosenblum, et. al., "The SimOS Simulation Environment", Stanford University, pp. 1-31, February 1997.

5. D. B. Hunt and P. N. Marinos, "A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique", Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems, pp. 170-175, 1987.

6. G. Richard III and M. Singhal, "Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory", Proceedings of the 12th Symposium on Reliable Distributed Systems, pp. 58-67, October 1993.

7. J. Singh, W. Weber, A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory*", technical report: csl-u-92-526, Stanford University, June 1992.

8. D. Sunada, D. Glasco, M. Flynn, "ABSS v2.0: a SPARC Simulator", Proceedings of the Eighth Workshou on Svnthesis and Svstem Integration of Mixed Technologies , pp. 143 - 149, October 1998.

9. D. Sunada, D. Glasco, M. Flynn, "Fault Tolerance: Methods Of Rollback Recovery", technical report: csl-tr-97-718, Stanford University, pp. l-51, March 1997.

10. D. Sunada, D. Glasco, M. Flynn, "Hardware-assisted Algorithms for Checkpoints", technical report: csl-tr-98-756, Stanford University, July 1998.

11. G. Suri, B. Janssens, et. al., "Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory", Proceedinas of the 25th International Svmposium on Fault-Tolerant Computing Systems, pp. 279-288, 1995.

12. S. C. Woo, M. Ohara, E. Torrie, J. Singh, A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", Proceedinas of the 22nd Annual International Svmposium on Computer Architecture, pp. 24 - 36, June 1995.

13. K. Wu, W. Fuchs, et. al., "Error Recovery in Shared Memory Multiprocessors Using Private Caches", "IEEE Transactions on Parallel and Distributed Systems", vol. 1, no. 2, pp. 23 l-240, April 1990.