Hardware-assisted Algorithms for Checkpoints

Dwight Sunada
David Glasco[1]
Michael Flynn

Technical Report: CSL-TR-98-756

July 1998

[1]Dr. David Glasco is affiliated with the Austin Research Laboratory at International Business Machines, Inc.

Hardware-assisted Algorithms for Checkpoints

Dwight Sunada
David Glasco
Michael Flynn

Technical Report: CSL-TR-98-756

July 1998

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

We can classify the algorithms for establishing checkpoints on distributed-shared-memory multiprocessors (DSMMs) into 3 broad classes: tightly synchronized method (TSM), loosely synchronized method (LSM), unsynchronized method (USM). TSM-type algorithms force the immediate establishment of a checkpoint whenever a dependency between 2 processors arises. LSM-type algorithms record this dependency and, hence, do not require the immediate establishment of a checkpoint if a dependency does arise; when a processor chooses to establish a checkpoint, the processor will query the dependency records to determine other processors that must also establish a checkpoint. USM-type algorithms allow a processor to establish a checkpoint without regard to any other processor. Within this framework, we developed 4 hardware-based algorithms: distributed recoverable shared memory (DRSM), DRSM for communication checkpoints (DRSM-C), DRSM with a hybrid method (DRSM-H), and DRSM with logs (DRSM-L). DRSM-C is a TSM-type algorithm, and DRSM and DRSM-H are LSM-type algorithms. DRSM-L is a USM-type algorithm and is the first of its kind for a tightly-coupled DSMM where hardware in the form of a directory maintains cache coherence. We find that DRSM has the best performance in terms of minimizing the impact of establishing checkpoints (or logs) on the running applications, but DRSM along with DRSM-C has the most expensive hardware requirements. DRSM-L has the second best performance but has the least expensive hardware requirement. We conclude that DRSM-L is the best algorithm in terms of cost and performance.

Key Words and Phrases: algorithm, checkpoint, fault tolerance, hardware, distributed recoverable shared memory (DRSM), DRSM for communication checkpoints (DRSM-C), DRSM with a hybrid method (DRSM-H), DRSM with logs (DRSM-L)

Copyright (c) 1998
Dwight Sunada, David Glasco, Michael Flynn

I. Introduction

The current direction of high-performance computing points towards distributed-shared-memory multiprocessors (DSMMs). Most of the research into these systems focuses on increasing the speed of the DSMM in executing applications. We focus on the different but equally significant issue of increasing the reliability of DSMMs while maintaining their high speed of execution. Reliability is the key to proliferating these systems in commercial businesses.

There are 2 categories of DSMMs: (1) collection of workstations loosely coupled by a general network on which specialized software maintains the coherence of memory accesses and (2) collection of processor boards tightly coupled by a dedicated network on which specialized hardware maintains the coherence of memory accesses. We select to investigate the second category of DSMMs, for they represent the highest level of performance that is available in computing.

Increasing reliability on such tightly-coupled DSMMs requires that we build fault tolerance into the system. Fault tolerance has 2 aspects: establishing checkpoints (or logging data) and rolling the system back from a fault to the last saved checkpoint. Establishing checkpoints has the greater impact on performance as this activity continuously consumes cycles even in a system that experiences no faults. Rolling the system back from a fault occurs rarely in well-built DSMMs (as they should be reliable) and hence has little impact on performance.

In our research, we classify checkpoint algorithms into 3 broad classes: tightly synchronized method (TSM), loosely synchronized method (LSM), and unsynchronized method (USM) [9]. In the TSM, a processor immediately establishes a checkpoint when a dependency on that processor arises. Examples of TSMs are algorithms for communication-induced checkpoints [1][5][13]. In the LSM, a processor need not immediately establish a checkpoint when a dependency arises, for the system simply records the dependency. Eventually, if a processor does establish a checkpoint (due to the expiration of a timer, for example), the processor looks at the records of dependencies to determine all other processors that must establish a checkpoint as well. An example of a LSM is the scheme proposed by Banatre [2]. In the USM, a processor establishes a checkpoint regardless of when any other processor establishes a checkpoint. The processor achieves this flexibility by logging the data retrieved from memory and re-using this data to satisfy cache misses during roll-back from a fault. To the best of our knowledge, algorithms for USM currently exist only for software-based loosely-coupled DSMMs [6][10].

In the remainder of this paper, we describe 4 hardware-based algorithms that we developed for establishing checkpoints. In section II, we describe the multiprocessor simulator (MPS) on which we simulate our hardware. In section III, we briefly discuss the virtual machine monitor (VMM). VMM is the piece of software which must be programmed to be aware of our hardware-based algorithms. In section IV, we present our assumptions about the hardware on which we propose to implement our 4 algorithms. In section V, we present 1 LSM-type algorithm: distributed recoverable shared memory (DRSM). In section VI, we present 1 TSM-type algorithm: DRSM for communication checkpoints (DRSM-C). Then, we attempt to improve on DRSM by removing 1 of the memory banks to create DRSM with a hybrid method (DRSM-H). We present DRSM-H in section VII. In section VIII, we present a USM-type algorithm: DRSM with logs (DRSM-L). DRSM-L is the first of its kind for USM on a hardware-based tightly-coupled

DSMM. To round out our discussion of the algorithms, we briefly discuss the issue of recovery in section IX. In section X, we discuss some issues concerning our implementation of these algorithms in our ABSS simulation environment. In section XI, we present the results of executing 6 benchmarks on our MPS modified for each of the algorithms. We conclude the paper in section XII.

Two principal aspects distinguish our work from previous work in this area. First, previous work uses rather different assumptions about the fault-tolerant hardware. For example, Wu proposes an algorithm for TSM and assumes that all the caches are fault tolerant, but Banatre assumes that the caches are not fault tolerant in his algorithm for LSM. Comparing 2 algorithms using markedly different hardware assumptions is questionable as the cost of those assumptions may cause a seemingly high-performance algorithm to be too expensive to implement. In our work, we make a uniform assumption across all the algorithms; namely, we assume only that the network and memory modules are fault tolerant but that the processor and its caches can fail. Second, we analyze in detail the performance of our algorithms by using a MPS that precisely simulates the operation of the cache and memory at the clock-cycle level.

II. Simulation Environment

A. Multiprocessor Simulator

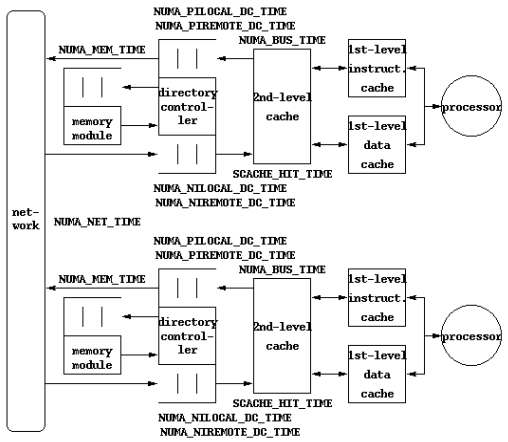

Figure 1. Multiprocessor

The MPS on which we implement the 4 algorithms is the SPARC-based multiprocessor simulator called ABSS [8]. It contains a thread module but does not have code to simulate either caches or memory. In order to simulate these structures, we extract the simulator of the memory system from SimOS [4] and write a C-language interface to hook the memory system into ABSS. Figure 1 illustrates the organization of the memory system. We specify the following parameters for the processor and memory system in our DSMM.

clock rate of processor = 200 megahertz for SPARC V7
size of 1st-level instruction cache = 32 kilobytes with 4-way set associativity, 64-byte line, and 2 states -- INVALID, SHARED
size of 1st-level data cache = 32 kilobytes with 4-way set associativity, 64-byte line, and 3 states--INVALID, SHARED, EXCLUSIVE
size of 2nd-level cache = 1 megabyte with 4-way set associativity, 128-byte line, and 3 states--INVALID, SHARED, EXCLUSIVE
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 to access (NUMA_MEM_TIME) memory = 50 cycles

Our DSMM has 8 modules, and each module has a SPARC processor, caches, and a memory module with a directory. On the processor, each integer instruction executes in exactly 1 cycle, and each floating-point instruction executes in a variable number of cycles. The DSMM implements a sequential memory model, where each access stalls until the memory system satisfies it. The cache-coherency protocol is a write-back protocol.

Although our simulator simulates 8 processors, we use a smaller DSMM to illustrate the description of our 4 hardware-assisted algorithms for establishing checkpoints. This smaller DSMM has 4 modules, where each module has a processor and a memory module with a directory.

B. Timing

Figure 1 shows the timing for the path which application-read and application-write accesses take. As an illustration, consider a read miss by processor 1 to a remote memory address for which processor 2 still holds the dirty copy of data; the memory address is remote to both processors. The read access misses in the 1st-level cache and progresses to the 2nd-level cache. Since it cannot satisfy the access, it incurs a delay of "NUMA_BUS_TIME" to travel on the local bus to the directory controller (DC). It places the access into the queue. When the access moves to the front of the queue, the DC uses a delay of "NUMA_PIREMOTE_DC_TIME" to process the access (i. e. to determine that it must travel to a remote memory module).

The DC sends the request across the network. The read access experiences a delay of "NUMA_NET_TIME" before arriving at the DC of the remote memory module. The remote DC places the access into the queue. When the access moves to the front of the queue, the DC uses a delay of "NUMA_NILOCAL_DC_TIME" to process the access. The DC sends a write-back request to processor 2. The write-back request uses a delay of "NUMA_NET_TIME" to travel to the DC of processor 2. The DC places the request into the queue. When the request moves to the front of the queue, the DC uses a delay of "NUMA_NILOCAL_DC_TIME" to process the request and forwards a copy of the request onto the local bus to the 2nd-level cache. The request uses a delay "NUMA_BUS_TIME" to travel to the cache and to return with the data dirty to the DC. It then sends an acknowledgment (containing the dirty data) of the write-back request across the network back to the DC of the remote memory module.

The acknowledgment uses a delay of "NUMA_NET_TIME" to arrive at the DC of the remote memory module. The remote DC places the acknowledgment into the queue. When the acknowledgment moves to the front of the queue, the DC uses a delay of "NUMA_NIREMOTE_DC_TIME" to process the acknowledgment. It contains a copy of the dirty data. The DC forwards a copy of the data to the memory and simultaneously sends a reply containing the data to processor 1. (The write-back request to memory waits in a queue. When the request moves to the front of the queue, the memory unit uses a delay of "NUMA_MEM_TIME" to write the dirty data into the specified memory address.)

The reply uses a delay of "NUMA_NET_TIME" to arrive at the DC of processor 1. The remote DC places the reply into the queue. When the reply moves to the front of the queue, the DC uses a delay of "NUMA_NIREMOTE_DC_TIME" to process the reply. The DC places the data contained in the reply onto the local bus. The data uses a delay of "NUMA_BUS_TIME" to travel across the bus to the 2nd-level cache.

C. Caches

The 1st-level data cache is a typical cache where each entry exists in 1 of 3 states: SHARED, EXCLUSIVE, and INVALID. ABSS handles a miss from the 1st-level cache in the following way. ABSS selects an unoccupied entry from the 1st-level miss-handling table and enters information about the miss into the entry. ABSS then submits the miss to the 2nd-level cache.

The 2nd-level unified cache is a typical cache where each entry exists in 1 of 3 states: SHARED, EXCLUSIVE, and INVALID. At the 2nd-level cache, if a miss occurs, ABSS selects an unoccupied entry from the 2nd-level miss-handling table and enters information about the miss into the entry. ABSS then submits the miss to the memory system.

D. Memory System

The memory system receives 5 types of messages.

1. MEMSYS_GET (read miss)
2. MEMSYS_GETX (write miss)
3. MEMSYS_UPGRADE (write hit on SHARED block in cache)
4. MEMSYS_WRITEBACK (forcing dirty block out of cache)
5. MEMSYS_REPLACEMENT_HINT (forcing clean block out of cache)

Application processes directly generate only the first 3 message types: "MEMSYS_GET", "MEMSYS_GETX", and "MEMSYS_UPGRADE". The 2nd-level cache itself may generate either MEMSYS_WRITEBACK or MEMSYS_REPLACEMENT_HINT to free an entry in the cache for incoming data.

III. Virtual Machine Monitor (VMM)

A. VMM and the Operating System (OS)

Establishing checkpoints requires that either the OS or the VMM cooperate with the underlying hardware. For example, communication between the DSMM and the external environment (e. g. a disk drive) requires that some (or all) of the processors in the DSMM establish a checkpoint. The OS or the VMM knows when this communication occurs and must manually tell the pertinent processors to establish a checkpoint.

We favor regarding the OS as simply another application process and building fault tolerance into the VMM. The VMM typically contains several thousands of lines of code whereas the OS typically contains millions of lines [3]. Hence, we can more easily modify the VMM. Since we separate the issue of fault tolerance from the OS, our fault-tolerant system can run any commodity OS without modification and still be fault tolerant.

B. VMM and ABSS

We neither build the VMM into ABSS nor simulate the communication between the DSMM and its environment. In an actual DSMM, this communication requires that the VMM manually initiate the establishment of a checkpoint on the pertinent processors. The activities involved in establishing this checkpoint are virtually identical to the activities involved in establishing a checkpoint required by the other triggers (identified in section V-C, section VI-B, section VII-B, and section VIII-B). So, understanding the performance impact of the reduced set of triggers should still enable us to understand the performance impact of the full set of triggers.

We focus on how our hardware-assisted algorithms (for establishing checkpoints) impact the performance of scientific application programs. In our simulation environment, we execute these programs directly on top of the raw simulated hardware. The application processes supply the memory references that drive the hardware-assisted algorithms for establishing checkpoints.

IV. Assumptions

We assume the following.

1. The hardware suffers at most a single point of failure.
2. Hardware is fail-stop. If a component fails, it merely stops functioning and does not emit spurious data.
3. Each memory module, including the on-board directory controller, is fault tolerant.
4. The processor board with its caches is not fault tolerant. One spare but idle processor board is available to replace another processor board that experiences a permanent fault.
5. A node in our DSMM can communicate with any other node via 2 independent paths in the network. A node consists of a memory module and a processor. (In the case of DRSM and DRSM-C, each memory module has 2 banks of memory.)
6. The VMM has fault containment and can coordinate with specialized hardware (for establishing checkpoints) to tell a processor to establish a checkpoint if its application process receives a message from the environment or sends a message to the environment. The VMM can also roll the DSMM back to the last checkpoint if a fault occurs.

Further, 2 identical banks of fault-tolerant memory are available. The number of memory modules equals the number of microprocessors, but within each memory module are 2 identical banks of memory.

Prior to each checkpoint, each block of memory has 2 copies, one being in each of the 2 banks of memory. One copy of the block receives new data sent to it, and the other copy is the value saved and frozen at the last checkpoint. In other words, 1 block is the tentative checkpoint, and 1 block is the permanent checkpoint. During the establishment of a checkpoint, the block with the tentative checkpoint freezes its data, which becomes the permanent checkpoint, and the block with the old checkpoint unfreezes its data, which becomes the tentative checkpoint. After the establishment of a checkpoint, the first access to the block with the tentative checkpoint causes the data in the permanent-checkpoint block to be copied to the tentative-checkpoint block.

We estimate that a single bank of fault-tolerant memory costs at most twice the price of a bank of non-fault-tolerant memory. So, 2 banks of fault-tolerant memory cost at most 4 times the price of a bank of non-fault-tolerant memory. Since the latter has a net cost of approximately 1/3 of the total system cost, we estimate that the 2 banks of fault-tolerant memory costs at most 2/3 of the total system cost.

V. Distributed Recoverable Shared Memory (DRSM)

DRSM is our first algorithm for establishing checkpoints and is a type of loosely synchronized method (LSM). We constructed DRSM by extending recoverable shared memory (RSM) to multiple memory modules. Banatre developed RSM; it uses a single memory module [2].

A. Background: Recoverable Shared Memory (RSM)

Before we describe DRSM, we first describe RSM. Figure 2 illustrates the configuration of a RSM module. We slightly modify the explanation by Banatre to present details that may not have appeared in the original description.

1. Dependency Matrix

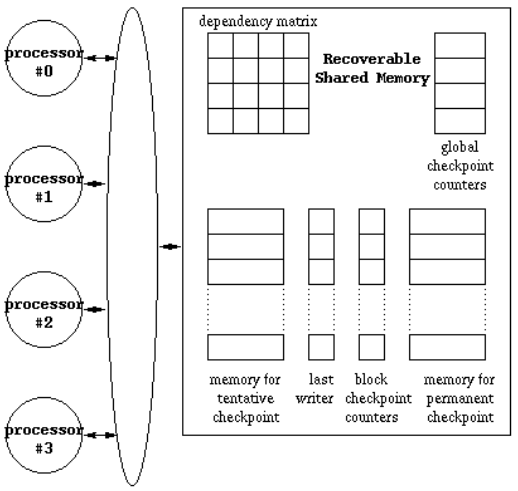

Figure 2. RSM

The key element is the dependency matrix, which is an array of bits. This matrix, "DM[][]", sees all accesses that reach the RSM module and sets the bits according to the following checkpoint dependencies.

write before read of block p[i] -> p[j]	(expression #1)
write before write of block p[i] <--> p[j]	(expression #2)

The write-before-read dependency occurs when processor "p[i]" reads a block (of memory) that processor "p[j]" previously wrote. The write before-write dependency occurs when processor "p[i]" writes a block that processor "p[j]" previously wrote.

In terms of checkpoints, the first dependency type means that if "p[i]" establishes a checkpoint, then "p[j]" must establish a checkpoint. The second dependency type means that if either "p[i]" or "p[j]" establishes a checkpoint, then the other processor must also establish a checkpoint in order to ensure the consistency of the state of the system stored in the checkpoint [9]. This information enables the processors to avoid the strict requirement of establishing a checkpoint at each communication between 2 processors where the first processor (in time) is a writer. (The algorithm for establishing checkpoints with DRSM-C does not record this information and hence strictly requires the establishment of a checkpoint at each communication where the first processor is a writer.)

2. Last-Writer Indicator

The RSM module contains an additional buffer for each block of memory. This buffer stores the last writer to the block.

3. Checkpoint Counters

The collection of global checkpoint counters contains 1 counter for each processor in the system. Each block in memory has an associated block checkpoint counter. When processor "P[2]", for example, writes to a block, the value of the global checkpoint counter for "P[2]" is copied into the block checkpoint counter. RSM updates the buffer for the last writer to "2". The purpose of the checkpoint counters is to accelerate the establishment of permanent checkpoints. They are explained in a later section.

4. Memory for Tentative Checkpoint

The tentative-checkpoint memory (TCM) functions as the working memory. It operates like the memory in a non-fault-tolerant system. When the RSM begins a 2-phase checkpoint of dependent processors, they write their dirty cache data into the TCM in the first phase. The newly updated blocks of memory form the tentative checkpoint which is later converted into the permanent checkpoint in the second phase.

5. Memory for Permanent Checkpoint

During the establishment of the permanent checkpoint, the RSM copies data saved during the tentative-checkpoint phase into the permanent-checkpoint memory (PCM). It always contains data comprising a consistent state of the system. The DSMM rolls back to this data after a failure occurs.

6. Establishing Checkpoints

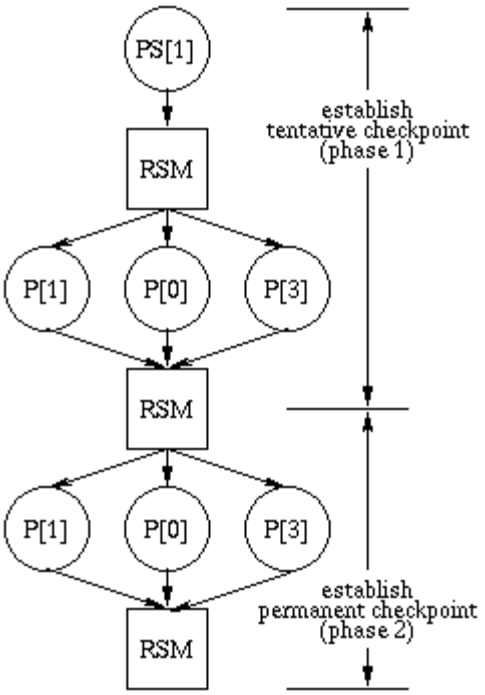

Figure 3. Flow of Checkpoint

Figure 3 illustrates the establishment of a checkpoint for a 4-processor DSMM. A processor "P[1]" that wishes to establish a checkpoint submits a request to the RSM. The RSM then scans the dependency matrix for all other processors that must establish a checkpoint along with "P[1]". The RSM finds that both "P[0]" and "P[3]" must establish a checkpoint along with "P[1]" and hence submits a request to them to establish a checkpoint. They write their dirty cache data back into memory and send a copy of their internal states (i. e. data in the internal registers) to the RSM. This phase is the tentative-checkpoint phase. After it completes successfully, the RSM converts the tentative checkpoint into a permanent checkpoint.

After verifying that the establishment of the tentative checkpoint is successful, the RSM increments the global checkpoint counters for "P[0]", "P[1]", and "P[3]" and then tells those processors that they may resume execution of their normal processes. They send acknowledgments to the RSM, and it resumes its normal functions. The RSM does not immediately copy the blocks saved during the tentative-checkpoint phase into the PCM during the permanent-checkpoint phase. Rather, the RSM merely increments the global checkpoint counters for the processors involved in the checkpoint. After the RSM concludes the permanent checkpoint, if a write access occurs on a block (in the TCM) where the global checkpoint counter of the last writer is greater than the block counter, then the RSM knows that the current data in the block is part of a permanent checkpoint. Hence, the RSM first copies the data from the block in the TCM into the corresponding block in the PCM before the RSM writes the incoming new data into the block in the TCM. This copy-on-write technique accelerates the establishment of the permanent checkpoint by avoiding the copying of

potentially millions of blocks of data from the TCM into the PCM during the permanent-checkpoint phase.

7. New Requests after Initiating Checkpoint

If processor "P[2]" submits a request to the RSM to establish a checkpoint and if the request arrives at the RSM before the start of the permanent checkpoint-phase for "P[0]", "P[1]", and "P[3]", then the RSM will combine "P[2]" into the group of processors that must establish a checkpoint together. In other words, the RSM grants the request from "P[2]" to establish a checkpoint and waits until "P[2]" has finished establishing its tentative-checkpoint phase before the RSM begins the permanent-checkpoint phase of all processors in the group: "P[0]", "P[1]", "P[2]", and "P[3]". If the request arrives after the establishment of the start of the permanent-checkpoint phase, then the RSM negatively acknowledges the request, and "P[2]" must re-submit its request.

B. Hardware Overview of DRSM

We extend RSM to multiple memory modules and call our method distributed-recoverable-shared memory (DRSM).

1. Elimination of Checkpoint Counters

DRSM eliminates the checkpoint counters that appear in figure 2. We no longer permanently fix the roles of the 2 banks of memory. A block in bank #1 and the corresponding block in bank #2 can be a tentative-checkpoint block and a permanent-checkpoint block, respectively, and vice versa. We use a 3-bit state register (3BSR) associated with each block of memory to dynamically determine the roles of the blocks in each pair of blocks, one from bank #1 and one from bank #2. Each 3BSR has its own tiny finite-state-machine (FSM) to cycle among the necessary states.

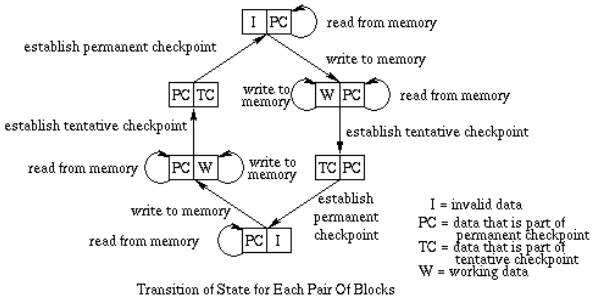

Figure 4. Transition of State for 3BSR

Figure 4 shows the state transitions (i. e. the role transitions) of the blocks in each pair of blocks of memory. The figure shows 6 rectangles; within each one are 2 smaller rectangles. Among these 2 rectangles, the one on the left represents a block from bank #1, and the rectangle on the right represents the corresponding block from bank #2. The states "[I, PC]" and "[PC, I]" are the 2 possible states in which memory starts.

As an illustration, we start with "[I, PC]". When a read access matches the pair of blocks with this state, DRSM services the access with data from bank #2. A write access (i. e. MEMSYS_GETX or MEMSYS_UPGRADE) that matches this pair of blocks causes the DRSM to copy data from the block in bank #2 into bank #1 and then to reply to the processor submitting the memory access. The state for the pair of blocks transitions to "[W, PC]". The DRSM services all future memory accesses by referencing the block in bank #1.

During the establishment of the tentative checkpoint, the state of the pair of blocks in our illustration transitions to "[TC, PC]". If the establishment of the tentative checkpoint is successful, then the DRSM changes the state of the pair of blocks to "[PC, I]".

2. New Organization of DSMM with DRSM

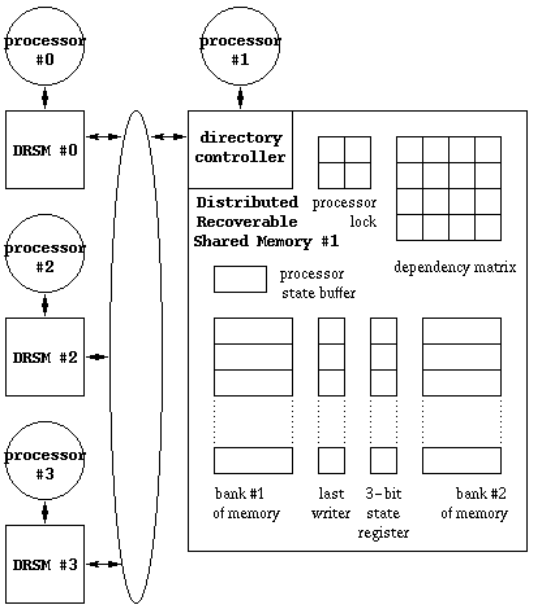

Figure 5. DRSM

Figure 5 shows the new organization of the DSMM with DRSM. Each DRSM module no longer contains the checkpoint counters. We replaced them with the more efficient 3BSRs. The DRSM module also has a processor state buffer (PSB). The PSB contains 3 buffers (not shown in figure 5). One buffer holds the internal state of the local processor for the tentative checkpoint, and another buffer holds the internal state of the local processor for the permanent checkpoint. The third buffer is a 2-bit checkpoint-state buffer (CSB) and holds one of {CHECKPOINT_IS_NOT_ACTIVE, TENTATIVE_CHECKPOINT_IS_ACTIVE, PERMANENT_CHECKPOINT_IS_ACTIVE}, indicating the status of the local processor.

The structure of processor locks (PLs) contains 1 lock per processor in the DSMM. The DRSM module sets the PL to "1" if, during the establishment of the tentative checkpoint, the corresponding processor (1) is the last writer of a dirty block (i. e. in state of "[PC, W]" or "[W, PC]") of memory or (2) has dependent processors according to the dependency matrix. During the establishment of the permanent checkpoint, if an incoming memory access originates from a processor with its PL being "1", the DRSM module negatively acknowledges that request. This action prevents a race from developing on the dependency matrix.

One example of a race is the following. Suppose that processor "P[1]" finishes its permanent checkpoint before a memory module "DRSM[2]" and that "P[1]" has its PL being "1" in "DRSM[2]". Suppose that "P[3]" writes into a memory block (in "DRSM[2]") that is not part of the checkpoint which is just completing. Then, processor "P[1]" reads that same block before "DRSM[2]" completes its permanent checkpoint. The dependency "P[1] -> P[3]" will be lost when "DRSM[2]" completes its permanent checkpoint, clearing the row and column (of the dependency matrix) containing "P[1]".

C. Conditions for Establishing Checkpoints

Two events can trigger a processor to establish a checkpoint.

1. A timer expires. When the timer for a processor expires, it establishes a checkpoint. The timer ensures a maximum bound on the time interval between checkpoints.
2. Communication occurs between a processor and the environment outside of the DSMM. When data leaves or enters a DSMM, the processor handling the data must establish a checkpoint. Communication includes interrupts.

D. Establishing Tentative Checkpoints

1. General Overview

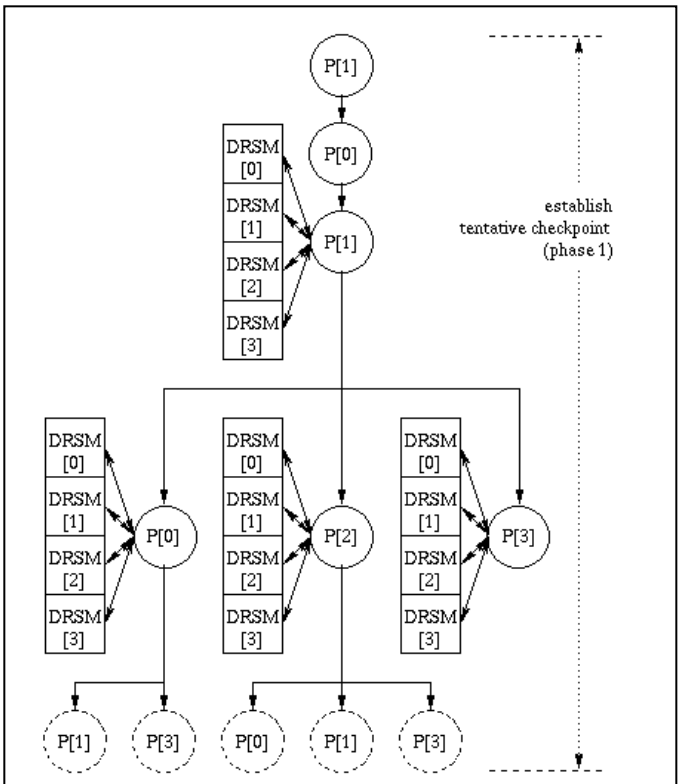

Figure 6. Flow of Tentative Checkpoint

Because there are now several modules of RSM, we must radically modify the algorithm for establishing a tentative checkpoint. Figure 6 shows the new algorithm. Its general strategy is that a processor wishing to establish a checkpoint must query all DRSM modules to determine all dependent processors that must also establish a checkpoint. They, in turn, query all DRSM modules to determine additional dependent processors. The process proceeds in the fashion of an expanding tree of processors. The root of the tree is the processor that initially wished to establish a checkpoint, and the leaves of the tree are processors that either (1) have no checkpoint-dependent processors or (2) have already been identified higher up in the tree. Once the algorithm reaches the leaves of the tree, processors starting from the bottom of the tree and moving upwards toward the root send acknowledgment messages to the parent processor. A parent processor must first receive acknowledgments from all its children before that parent processor sends an acknowledgment to its parent processor.

2. Details

As a specific example, we trace the flow in figure 6 for a 4-processor DSMM. We arbitrarily select processor "P[0]" to act as an arbiter to allow at most one tentative checkpoint to be established at any time. "P[1]" submits a request to "P[0]" to obtain permission to establish a checkpoint. "P[0]" grants the request, and "P[1]" proceeds to establish a tentative checkpoint (in the first phase).

"P[1]" queries all the DRSM modules. They search their dependency matrices to find all processors which must establish a checkpoint along with "P[1]". The DRSM modules send their replies back to "P[1]". From their replies, "P[1]" discovers that "P[0]", "P[2]", and "P[3]" must establish checkpoints. "P[1]" tells all of them to establish tentative checkpoints. In this example, "P[0]", "P[2]", and "P[3]" receive the request from "P[0]" at approximately the same time (although this situation need not always arise).

"P[0]" then queries all the DRSM modules to find that both "P[1]" and "P[3]" are dependent upon it. "P[0]" asks them to establish tentative checkpoints. Upon receiving this request from "P[0]", "P[1]", and "P[3]" both respond that they have already joined the checkpoint tree and are, hence, leaves in this tree.

After receiving the request from "P[1], "P[2]" then queries all the DRSM modules to find that "P[0]", "P[1]", and "P[3]" are dependent upon it. "P[2]" asks them to establish tentative checkpoints. Upon receiving this request from "P[2]", "P[0]", "P[1]", and "P[3]" all respond that they have already joined the checkpoint tree and are, hence, leaves in this tree.

Finally, after receiving the request from "P[1]", "P[3]" then queries all the DRSM modules to find that no processors are dependent on it. Hence, "P[3]" is a leaf in this tree. After "P[3]" completes its tentative checkpoint, "P[3]" sends an acknowledgment back to "P[1]".

After "P[0]" completes its tentative checkpoint, "P[0]" sends an acknowledgment back to "P[1]". After "P[2]" completes its tentative checkpoint, "P[2]" sends an acknowledgment back to "P[1]". After "P[1]" receives acknowledgments from "P[0]", "P[2]", and "P[3]", "P[1]" concludes the establishment of the tentative checkpoint, which is phase #1.

Figure 6 shows the general flow in the process of establishing a tentative checkpoint but omits 6 important details. They are the following.

1. Before a processor queries all DRSM modules (in order to determine dependent processors), it (1) waits until all its pending cache operations are finished and (2) then writes all dirty cache data back into memory. The processor waits for the DRSM modules to acknowledge that all the write-backs are complete.
2. Each processor sends a copy of its state (i. e. data in the internal registers) to the PSB of the DRSM module that is local to the processor.
3. A DRSM module that receives a query (to determine dependent processors) waits until all pending memory operations by the directory controller are finished before the DRSM module replies (with information about dependent processors) to the querying processor. During this waiting period, the DRSM module negatively acknowledges all requests that it receives.
4. Just before the DRSM module replies to the querying processor, the DRSM module scans for all pairs of blocks (in the 2 banks of memory) where (1) the state of the pair is either "[W, PC]" or "[PC, W]" and (2) the last writer is the querying processor. If such pairs exist, then the DRSM module transitions the states from "[W, PC]" or "[PC, W]" to "[TC, PC]" or "[PC, TC]", respectively. The DRSM module negatively acknowledges accesses to blocks for which the state is either "[TC, PC]" or "[PC, TC]". Locking out accesses to such blocks prevents changes in the checkpoint dependencies of the processors that have almost completed the tentative checkpoint.
5. In addition, the DRSM module sets the PL of the querying processor to "1" if (1) the module has dirty blocks like those just mentioned or (2) the dependency matrix indicates that the querying processor has dependent processors. The DRSM module negatively acknowledges normal memory accesses originating from a processor with its PL being "1". The aim is to prevent a race condition from developing in the dependency matrix when the DRSM module finishes its permanent checkpoint after a processor has finished its checkpoint.
6. Just before the root processor of the checkpoint tree begins the establishment of the tentative checkpoint, that processor sets a 2-bit checkpoint-state buffer (CSB) in the PSB of the local memory module to indicate that the establishment of the tentative checkpoint is active. The state of the CSB can be one of {CHECKPOINT_IS_NOT_ACTIVE, TENTATIVE_CHECKPOINT_IS_ACTIVE, PERMANENT_CHECKPOINT_IS_ACTIVE}. In figure 6, processor "P[1]" sets the CSB to "TENTATIVE_CHECKPOINT_IS_ACTIVE". The DSMM uses this state information to determine what to do in the event that a fault occurs during the establishment of a checkpoint.

3. Dependent Processors and Dependent DRSM Modules

The acknowledgments that are propagated from the leaves of the checkpoint tree up to the root in figure 6 lists the dependent processors. Each processor in the tree determines processors that are checkpoint dependent upon itself, packages this information along with all dependent-processor information from the child processors, passes this package of information in an acknowledgment to the parent processor. In the end, the root processor is aware of all dependent processors in the entire tree.

The tree also propagates information about dependent DRSM modules to the root processors. A DRSM module is a dependent DRSM module if any of the following conditions is true.

1. The DRSM module identifies at least one processor that is dependent on the querying processor during the tentative-checkpoint phase.
2. The DRSM module identifies the querying processor as one that has written to the module since the last checkpoint. The DRSM module can determine this condition by finding one pair of blocks (in the 2 banks of memory) for which (1) the last writer is the querying processor and (2) the data in one block of the pair is in state "W" (i. e. working data).

When the DRSM module replies (with information about the dependent processors) to the querying processor, the DRSM module also tells it whether the module is a dependent memory module. The querying processor propagates this information about dependent DRSM modules back up to the root processor of the checkpoint tree.

E. Establishing Permanent Checkpoints

1. General Overview

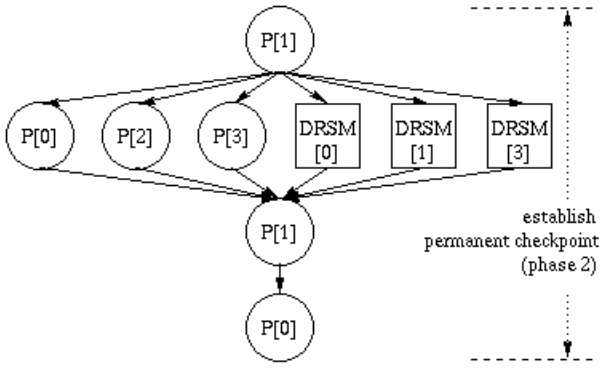

Figure 7. Flow of Permanent Checkpoint

Figure 7 shows the new algorithm for establishing a permanent checkpoint for a 4-processor DSMM. The general strategy is that the root processor in the checkpoint tree for the tentative checkpoint guides the establishment of the permanent checkpoint. The root processor tells all dependent processors and all dependent DRSM modules to establish a permanent checkpoint. Once they complete their permanent checkpoints, they send acknowledgments to the root processor. It then completes its own permanent checkpoint and sends an acknowledgment to the arbiter processor. It removes the root processor from the queue.

2. Details

As a specific illustration, we trace the flow in figure 7. "P[1]" is the root processor of the checkpoint tree for the tentative checkpoint. From it, "P[1]" knows that "P[0]", "P[1]", and "P[3] are dependent processors and that "DRSM[0]", "DRSM[1]" , and "DRSM[3]" are dependent DRSM modules. In this

particular example, "DRSM[2]" is not a dependent DRSM module. "P[1]" tells all dependent processors and DRSM modules to establish a permanent checkpoint. Each of "DRSM[0]", "DRSM[1]", and "DRSM[3]" performs the following. The DRSM module resets (to zero) all columns and all rows (in the dependency matrix) containing any processor with its PL being "1". Next, the DRSM module identifies all pairs of blocks (in the 2 banks of memory) for which their states are "[TC, PC]" or "[PC, TC]". The DRSM module transitions their states to "[PC, I]" or "[I, PC]", respectively. Finally, the DRSM module sends an acknowledgment to the root processor.

After receiving the request to establish a permanent checkpoint, each of "P[0]", "P[1]", and "P[3]" sends an acknowledgment back to the root processor and resumes normal processing. After "P[1]" receives acknowledgments from all dependent processors and memories, "P[1]" itself sends an acknowledgment to the arbiter processor "P[0]" and resumes normal processing. "P[0]" then grants the request from the next processor wishing to establish a checkpoint.

We should note the following additional details. "P[1]" sets the CSB to "PERMANENT_CHECKPOINT_IS_ACTIVE" just before "P[1]" begins the phase for the establishment of the permanent checkpoint. The dependent DRSM modules clear all the PLs to "0" during phase 2. After the establishment of the permanent checkpoint is complete, "P[1]" sets the CSB to "CHECKPOINT_IS_NOT_ACTIVE". Also, each processor tells its local memory module to designate the tentative checkpoint of the processor state in the PSB as a permanent checkpoint. The processor does not wait for an acknowledgment from the local memory module before sending an acknowledgment to "P[1]" since the network is reliable by virtual of our assuming 2 independent paths between any two nodes.

F. Additional Features

1. Artificially Dependent Processors

If the establishment of a checkpoint is in progress, the arbiter queues requests from processors that request permission to establish a checkpoint. After the arbiter receives an acknowledgment that the establishment of the current checkpoint is complete, the arbiter grants the next processor (waiting in the checkpoint-request queue) permission to establish a checkpoint. There can be many processors waiting in the queue.

If more than 1 processor waits in the queue, the arbiter grants permission to the processor at the front of the queue but tells that processor, say "P[1]", to artificially treat all the other processors (in the queue) as being dependent on it. After "P[1]" finishes querying the DRSM modules to find the genuinely dependent processors, "P[1]" adds the artificially dependent processors to this group of genuinely dependent processors. Then, "P[1]" requests that all of them establish tentative checkpoints.

In this way, any processor that submits a request to establish a checkpoint to the arbiter processor needs to wait at most approximately the period for establishing one checkpoint. That one checkpoint is the one that is currently being established when the request arrives at the arbiter.

2. Arbiter

The algorithm for the DRSM uses an arbiter, which is "P[0]" in our example. Although it appears to be a potential bottleneck in the DSMM, the arbiter actually poses no particular problem. The maximum number of requests (for the

establishment of checkpoints) that can queue at the arbiter is "(number of processors in the DSMM) - 1". After the current establishment of a checkpoint completes, the arbiter tells the processor, say "P[2]", of the first request in the queue to begin the establishment of a checkpoint. If there are other requests in the queue, the arbiter tells "P[2]" to include their corresponding processors as artificially dependent processors. In other words, after the current establishment of a checkpoint completes, all processors with requests waiting in the queue participate in the next establishment of a checkpoint, clearing the entire queue. The maximum delay between (1) the arrival of a request at the arbiter and (2) the participation (in the establishment of a checkpoint) by the processor submitting the request is approximately the time required for the current establishment of a checkpoint to complete.

G. Summary

The DRSM (and the RSM) basically records dependencies that arise among processors as they access the same memory locations. Recording the dependencies generally enables the DRSM to delay the establishment of checkpoints until an arbitrarily chosen time. In our case, we use a timer to announce when a processor should establish a checkpoint; the timer can set the maximum temporal interval between checkpoints, effectively setting the maximum time for roll-back recovery. Any interaction between an application process and the environment of the DSMM poses a special problem and requires that the processor of the application process must immediately establish a checkpoint.

VI. Distributed Recoverable Shared Memory for Communication Checkpoints (DRSM-C)

Distributed recoverable shared memory for communication checkpoints is our second algorithm for establishing checkpoints. It is a tightly synchronized method (TSM). Wu presents one version of it [13]. The principal difference between the TSM and the LSM is that the former forces the establishment of a checkpoint for a processor if that processor become checkpoint dependent on another processor.

In the version described by Wu, a processor "P[3]" must establish a checkpoint if "P[1]" reads from data or writes to data that is cached in the dirty state in "P[3]". In addition, "P[3]" must establish a checkpoint if it writes dirty cache data back into main memory.

A. Hardware Overview

We apply the TSM to our base DSMM. Our system differs from that of Wu in 2 aspects. Wu uses a DSMM with both fault-tolerant caches and exactly 1 bank of fault-tolerant memory. By contrast, our DSMM has caches that are not fault-tolerant, but our system does have 2 banks of fault-tolerant memory.

The 2 banks enable us to eliminate 1 cause of establishing checkpoints. Namely, a processor can write dirty cache data back into main memory without requiring the establishment of a checkpoint. The DSMM must still establish a checkpoint for a processor, say "P[3]", whenever the system transfers dirty data written by "P[3]" to another processor, say "P[1]". This dirty data need not reside in "P[3]" at the moment of the transfer but could reside solely in main memory.

Figure 8 illustrates the distributed-recoverable-shared-memory-for-communication-checkpoint (DRSM-C) module. The 3-bit state register (3BSR) and

the last-writer buffer for each block indicate, respectively, (1) whether data in a block is dirty and (2) which processor must establish a checkpoint in the event that the dirty data is transferred to another processor. The DRSM-C module omits the dependency matrix. Any dependency that arises immediately forces the establishment of a checkpoint; the newly established checkpoint erases the dependency and hence obviates the need for a matrix to record the dependency.

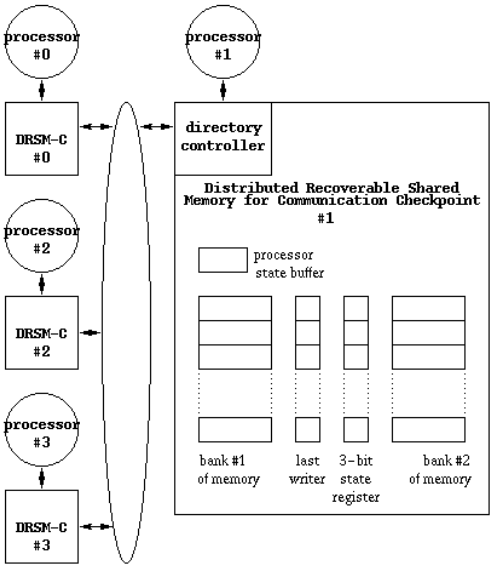

Figure 8. DRSM-C

When an access arrives at a DRSM-C module, it checks whether the matching block of memory contains dirty data. If the block contains dirty data, the DRSM-C module sends a negative acknowledgment to the memory-accessing processor and then requests that the last-writer processor (of the block with dirty data) establishes a checkpoint. The 3-bit register and the last-writer buffer provide adequate information to determine whether to negatively acknowledge a memory access.

B. Conditions for Establishing Checkpoints

Three events can trigger a processor to establish a checkpoint.

1. A timer expires. When the timer for a processor expires, it establishes a checkpoint. The timer ensures a maximum bound on the time interval between checkpoints.
2. Communication occurs between a processor and the environment outside of the DSMM. When data leaves or enters a DSMM, the processor handling the data must establish a checkpoint. Communication includes interrupts.
3. Dirty data is transferred between processors. Since the TSM does not record dependencies among processors and hence does not permit roll-back propagation, any dependency that arises (like that caused by transferring dirty data between processors) forces the establishment of a checkpoint to erase the dependency.

C. Establishing Checkpoints

1. General Overview

Figure 9 illustrates the algorithm for establishing a checkpoint for a 4-processor DSMM. The general strategy is that a processor wishing to establish a checkpoint simply does the following. The processor writes its dirty cache data back into memory and requests that each DRSM-C module establishes a tentative checkpoint for the blocks containing data written by the processor. Once all the modules acknowledge completion of the tentative checkpoint, the processor requests that each module convert the tentative checkpoint into a permanent checkpoint. Once the modules tell the processor that the establishment of the permanent checkpoint is complete, the processor finishes the establishment of the checkpoint and resumes normal processing.

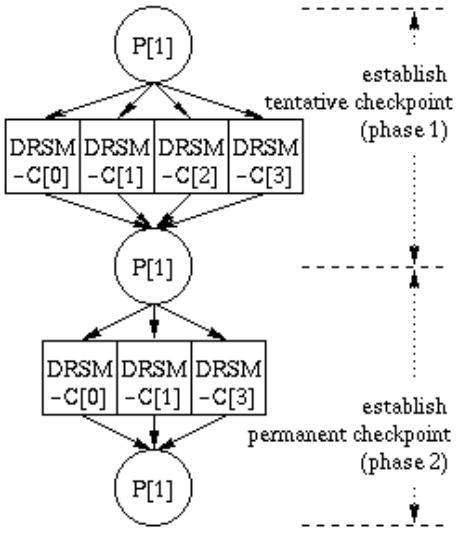

Figure 9. Flow of Checkpoint

2. Details

As a specific illustration, we trace the flow in figure 9. "P[1]" begins the establishment of the tentative checkpoint. "P[1]" requests that each DRSM-C module establishes a tentative checkpoint for each pair of blocks (in both banks of memory) that have been written by "P[1]". Once the modules finish the tentative checkpoint, they send acknowledgments to "P[1]". It then begins the second phase, which is establishing the permanent checkpoint. "P[1]" then requests that each DRSM-C module except "DRSM-C[2]" transitions each pair of blocks in the tentative-checkpoint state into the permanent-checkpoint state. In our example, "DRSM-C[2]" does not contain any blocks to which "P[1]" has written data since its last checkpoint, so "DRSM-C[2]" does not transition any pair of blocks into the tentative-checkpoint state and hence does not participate in the permanent-checkpoint phase. After all 3 DRSM-C modules complete the transition of blocks into the permanent-checkpoint state, they send acknowledgments to "P[1]". It then resumes normal processing.

Figure 9 shows the general flow in establishing a tentative checkpoint but omits 3 important details. They are the following.

1. Before a processor requests all DRSM-C modules to transition blocks into the tentative-checkpoint state, the processor (1) waits until all its pending cache operations are finished and (2) then writes all dirty cache data back into memory. The processor waits for the DRSM-C modules to acknowledge that all the write-backs are complete.
2. Each processor sends a copy of its state (i. e. data in the internal registers) to the PSB of the DRSM-C module that is local to the processor.
3. Just before the processor begins the establishment of the tentative checkpoint, that processor sets a 2-bit checkpoint-state buffer (CSB) in the PSB of the local memory module to indicate that the establishment of the tentative checkpoint is active. The state of the buffer can be one of {CHECKPOINT_IS_NOT_ACTIVE, TENTATIVE_CHECKPOINT_IS_ACTIVE, PERMANENT_CHECKPOINT_IS_ACTIVE}. In figure 9, processor "P[1]" sets the CSB to "TENTATIVE_CHECKPOINT_IS_ACTIVE". The DSMM uses this state information to determine what to do in the event that a fault occurs during the establishment of a checkpoint.

Also, we should note the following. "P[1]" sets the CSB to "PERMANENT_CHECKPOINT_IS_ACTIVE" just before "P[1]" begins the phase for the establishment of the permanent checkpoint. After its establishment is complete, "P[1]" sets the CSB to "CHECKPOINT_IS_NOT_ACTIVE". Also, each processor tells its local memory module to designate the tentative checkpoint of the processor state in the PSB as a permanent checkpoint.

D. Summary

Unlike the DRSM, the DRSM-C does not record dependencies that arise among processors as they access the same memory locations. Hence, the DRSM-C requires that a processor immediately establish a checkpoint if the processor

reads or writes a block (of memory) to which another processor has already written data. In addition, to establish a maximum bound on the temporal interval between the checkpoints for a processor, we use a timer to announce when the processor must establish a checkpoint. Any interaction between an application process and the environment of the DSMM also causes the establishment of a checkpoint.

VII. Distributed Recoverable Shared Memory with a Hybrid Method (DRSM-H)

A. Hardware Overview

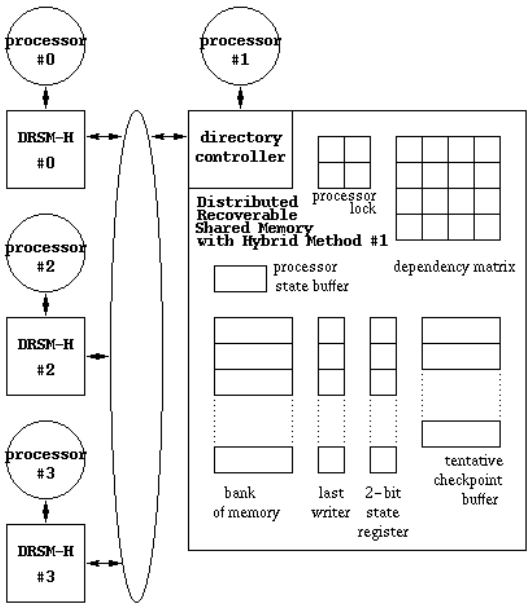

Figure 10. DRSM-H

Figure 10 illustrates the arrangement of the DRSM-H. In designing it, we seek to maintain the good performance of DRSM but to eliminate the cost of the second bank of memory. Hence, the DRSM-H omits the second bank of memory and restricts the remaining bank of memory to always hold the permanent checkpoint. During the establishment of a tentative checkpoint, each processor involved in it writes the dirty cache lines into a new buffer, the tentative checkpoint buffer (TCB); during the establishment of the permanent checkpoint, each processor involved in it writes the dirty cache lines back into main memory. The TCB ensures that the DRSM-H can recover from a fault even if it occurs during the establishment of the permanent checkpoint.

Since the DRSM-H has only 1 bank of memory for the permanent checkpoint, whenever the 2nd-level cache must write dirty data back into main memory due to a conflict or capacity miss, the processor of that cache must establish a permanent checkpoint. In order to keep dirty data "floating" among the caches as long as possible before it is written back into main memory, we modify the 2nd-level cache to use 4 states: INVALID, SHARED, DIRTY_SHARED, and EXCLUSIVE. Once a block enters the state of EXCLUSIVE, the block changes state among DIRTY_SHARED and EXCLUSIVE.

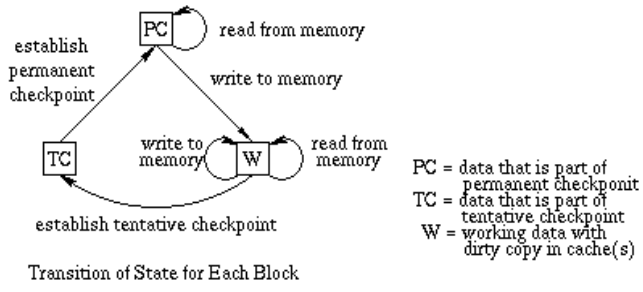

Figure 11. Transition of 2BSR

Also, DRSM-H replaces the 3BSR with a 2-bit state register (2BSR). For each block, the 2BSR transitions among the states indicated in figure 11. When a 2nd-level cache issues an EXCLUSIVE access to a block in the state of "PC", the 2BSR transitions from "PC" to "W".

Compared to the dependency matrix in DRSM, the dependency matrix in DRSM-H records the

following more stringent conditions for dependency.

read after write of block	
P[i] <--> P[j]	(expression #3)
write after write of block	
P[i] <--> P[j]	(expression #4)

The read-after-write case now becomes a 2-way dependency. If the dependency matrix recorded dependencies according to expression #1 mentioned earlier, the following situation can arise. Processor "P1" writes data into memory block "BA". Then "P2" reads the value of that block, which resides in state DIRTY_SHARED, and "P1" subsequently evicts "BA" from the 2nd-level cache. Next, "P1" establishes a checkpoint and must write the value in "BA" into main memory. Unfortunately, "P1" cannot easily find "BA" since (1) it is no longer in the cache of "P1" and (2) no checkpoint dependency (according to expression #1) exists between "P1" and "P2". Hence, to solve this problem in a simple way, we replace expression #1 with expression #3.

B. Conditions for Establishing Checkpoints

A processor in a DSMM with DRSM-H establishes a checkpoint when any of the following 3 conditions arises.

1. A timer expires. The timer determines the maximum temporal interval between checkpoints.
2. The 2nd-level cache (1) evicts a cache line in state EXCLUSIVE without forwarding the line to another cache or (2) evicts the last copy of a cache line in state DIRTY_SHARED. Both events require that a DRSM-H module write the dirty line back into memory, which holds the `permanent` checkpoint.
3. Communication occurs between the processor and the environment outside of the DSMM.

C. Establishing Tentative Checkpoints

The procedure for establishing a tentative checkpoint is similar to that illustrated in figure 6 for a 4-processor DSMM with DRSM. The principal difference between the procedure for DRSM-H and that for DRSM is that the processor in the DRSM-H does not write the dirty 2nd-level-cache lines back into main memory. Rather, the processor writes the dirty lines into the TCB. During the establishment of the permanent checkpoint, the processor writes the dirty lines back into main memory.

D. Establishing Permanent Checkpoints

1. General Overview

Figure 12 shows the new algorithm for establishing a permanent checkpoint for a 4-processor DSMM, following from the events in figure 6. The algorithm is similar to that illustrated in figure 7 for DRSM. The principal difference is that the dependent processors must first write their dirty 2nd-level-cache lines back into main memory and complete the permanent checkpoints of the processors before the dependent DRSM-H modules can complete their permanent checkpoints. The root processor, "P[1]" writes its dirty 2nd-level-cache lines back into main memory but does not complete the permanent checkpoint until the DRSM-H modules complete their permanent checkpoints.

2. Details

As a specific illustration, we trace the flow in Figure 12. During phase 1, "P[1]" determined that there are 3 dependent processors -- "P[0]", "P[2]", and "P[3]" -- and 3 dependent memory modules -- "DRSM-H[0]", "DRSM-H[1]", and "DRSM-H[3]". In this particular example, "DRSM[2]" is not a dependent DRSM-H module. "P[1]" tells all dependent processors to establish a permanent checkpoint.

After receiving the request to establish a permanent checkpoint, each of "P[0]", "P[1]", and "P[3]" writes its dirty 2nd-level-cache lines back into main memory. The processor waits until the write-back is complete, then sends an acknowledgment back to the root processor, and resumes normal processing. After "P[1]" receives acknowledgments from all dependent processors, "P[1]" tells all dependent DRSM-H modules to establish a permanent checkpoint.

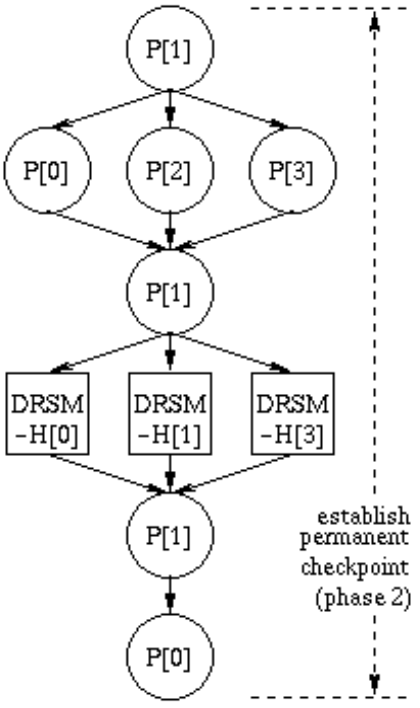

Figure 12. Flow of Permanent Checkpoint

Each of "DRSM-H[0]", "DRSM-H[1]", and "DRSM-H[3]" performs the following. The DRSM-H module resets (to zero) all columns and all rows (in the dependency matrix) containing any processor with its PL being "1". Next, the DRSM-H module identifies all blocks for which their states are "[TC]". The DRSM-H module transitions their states to "[PC]". Finally, the DRSM-H module sends an acknowledgment to the root processor.

After "P[1]" receives acknowledgments from all and memories, "P[1]" itself sends an acknowledgment to the arbiter processor "P[0]" and resumes normal processing. "P[0]" then grants the request from the next processor wishing to establish a checkpoint.

Unlike the dependent memory modules in DRSM, the ones in DRSM-H must establish a permanent checkpoint after the dependent processors establish a permanent checkpoint. The dependent processors must write the dirty blocks in their caches back into main memory (which contains the permanent checkpoint) during the establishment of the permanent checkpoint. Only after this activity is complete can the dependent memory modules convert their tentative checkpoint into a permanent one.

Concerning figure 12, we note the following additional details for DRSM-H. At the start of phase 2, "P[1]" updates the checkpoint-state buffer (CSB) of the PSB to "TENTATIVE_CHECKPOINT_IS_ACTIVE". During phase 2, the dependent processors write all the dirty blocks in their 2nd-level caches back into main memory. A safe copy of these blocks exists in the TCB, so if a fault occurs during the write-back, the DSMM can still recover from the fault. Further, each dependent processor directs the PSB to invalidate its old permanent

checkpoint and to designate the processor state saved in the tentative-checkpoint area as the new permanent checkpoint. Each dependent memory module clears, in the dependency matrix, all rows and columns containing any processor with its PL being "1" and then resets all the PLs to "0". At the end of phase 2, "P[1]" directs the PSB to invalidate its old permanent checkpoint and to designate the processor state saved in the tentative-checkpoint area as the new permanent checkpoint, and "P[1]" then updates the CSB of the PSB to "CHECKPOINT_IS_NOT_ACTIVE", indicating that phase 2 (and the entire checkpoint) is complete.

E. Summary

DRSM-H is similar to DRSM but omits 1 of the 2 banks of memory. The remaining bank of memory always holds the permanent checkpoint. So, we must expand the number of triggers causing the establishment of a checkpoint to include the event where a dirty 2nd-level-cache line is written back into main memory. In order to maximally delay when a dirty line is written back into memory, we try to keep the dirty line "floating" among the caches as long as possible; towards that aim, we increase the number of cache-line states to include the state of DIRTY_SHARED.

A processor in the DSMM with DRSM-H writes a dirty 2nd-level-cache line back into memory only during the estabishment of the permanent checkpoint. In order to ensure that the system can recover from a fault that occurs during this write-back, the processor first writes all dirty cache lines into the TCB during the establishment of the tentative checkpoint. If a fault occurs during the actual write-back into main memory, the DSMM can complete the write-back by retrieving the dirty lines from the TCB.

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

Several USM-type algorithms for establishing checkpoints in loosely-coupled DSMMs (where software maintains the coherence of memory accesses) exist [6][10]. These algorithms are all implemented in software. DRSM-L represents the first attempt at constructing a hardware-based USM-type algorithm that establishes checkpoints for tightly-coupled DSMMs.

A. Hardware Overview

Figure 13 illustrates the arrangement of the DRSM-L. It extends each line in the 2nd-level cache to include a counter, a write-after-total-log flag (WATLF), and a status-of-eviction flag (SEF). The counter counts the number of accesses that hit in a 1st-level cache line. If a local access hits in the 1st-level cache, it simultaneously (1) returns the data in the matching line to the processor and (2) forwards a hit notice to the 2nd-level cache so that it can increment the counter. The 2nd-level cache also contains a queue to hold incoming accesses from remote caches when the local cache is stalled during the logging of 2nd-level-cache lines into the local memory module.

The DRSM-L module contains a line buffer and a counter buffer, of which both function as logs. The 2nd-level cache fills the line buffer with cache lines and fills the counter buffer with cache counters. The cache also copies the extended tags into the buffers; an extended tag is a normal cache tag appended with the index of the cache line. Also, each entry in the counter buffer contains a status-of-eviction flag (SEF); the DRSM-L sets it to "1" if the counter comes from a cache line that the cache will evict. Each of the line buffer and the counter buffer has its own index register that points to the

next free entry in the buffer. After the local processor establishes a checkpoint, the processor resets both index registers to "0", effectively clearing both buffers. Also, the memory module contains both a processor state buffer (PSB) and a tentative checkpoint buffer (TCB).

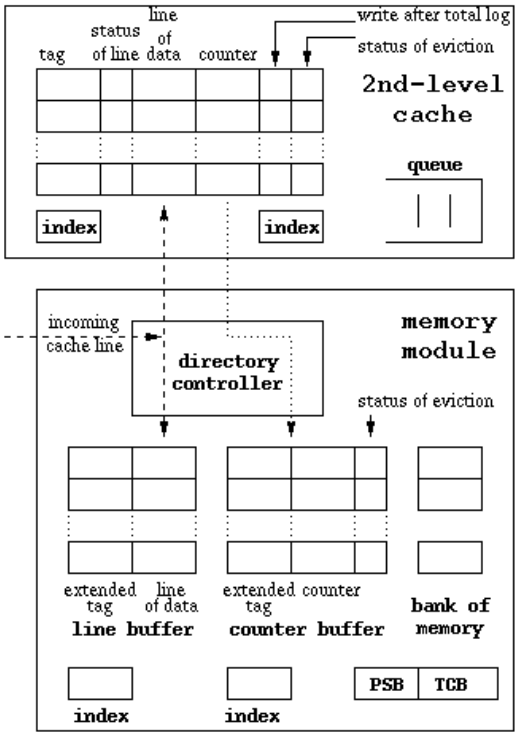

Figure 13. DRSM-L

The 2nd-level cache maintains 2 index registers that are mirror images of the index registers in the DRSM-L module. Before the cache copies a line or a counter (or group of counters) into the line buffer or counter buffer, respectively, the cache checks its appropriate on-board index register. If the copy will cause an overflow in either the line buffer or the counter buffer, the cache will initiate the establishment of a checkpoint on the processor in order to reset all the index registers to "0". If the copy will not cause an overflow, then the cache copies a line or a counter (or group of counters) into the line buffer or counter buffer, respectively, and the cache increments its appropriate on-board index register.

B. Conditions for Establishing Checkpoints

A processor in a DSMM with DRSM-L establishes a checkpoint when any of the following 4 conditions arises.

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 DSMM.

C. Establishing Logs

1. Motivation

DRSM-L enables each processor to establish a checkpoint without regard to any other processor. The DRSM-L algorithm achieves this flexibility by recording, in a log, the value of each new memory block arriving in the 2nd-level cache. If a transient fault on a processor occurs, the recovery algorithm (1) resets it, (2) resumes its execution from the processor state saved in the last checkpoint, and then (3) uses the stored values in the log to satisfy 2nd-level-cache misses during the recovery. After the processor consumes all the values in the log, the processor has returned to the state just prior to the occurrence of the fault, and the recovery is finished. The net effect is that the value in any dirty cache line sent to a remote cache is never lost. The log contains the values of the cache lines that the recovering local processor

needs to re-compute (and reproduce) exactly all the old values that it wrote into its dirty cache lines. Hence, dependencies among processors never arise, and each of them can establish a checkpoint without regard to any other processor.

2. Detailed Discussion

Each DRSM-L module contains 2 logs: line buffer and the counter buffer. Whenever an incoming cache line arrives at the DRSM-L module, it concurrently (1) writes the extended tag (2nd-level-cache tag plus index) and data for that line into the line buffer and increments the index to point at the next free entry and (2) forwards the line to the 2nd-level cache. Since these 2 activities occur concurrently, the delay for the second activity hides the delay for the first activity. The module logs only incoming cache lines that satisfy data misses and ignores lines that satisfy instruction misses. When the local processor accesses a line in the 1st-level cache, it simultaneously (1) sends the data in the line to the processor and (2) forwards a hit notice to the 2nd-level cache to increment its counter. Again, the delay for the first activity hides the delay for the second activity. If the hit notice is for a write, then the cache sets the WATLF in addition to incrementing the counter. If the counter overflows, the cache copies both the extended tag (of the cache line) and its counter into the counter buffer and sets the SEF in the memory module to "0". The counter is 32-bits wide.

Eventually, an access from a remote cache via a directory controller arrives at the local 2nd-level cache and hits in a valid cache line, or the local 2nd-level cache evicts a cache line due to a conflict (or capacity) miss. Three possibilities arise in our simulated DSMM from the perspective of the local 2nd-level cache.

1. event = eviction (due to conflict miss or remote invalidation) and cache-line state = SHARED
2. event = flush (due to remote read) and cache-line state = EXCLUSIVE
3. event = eviction (due to conflict miss or remote invalidation) and cache-line state = EXCLUSIVE

In case #1, the cache copies the extended tag and the counter of the cache line into the counter buffer and sets the SEF in the memory module to "1" since the line will be evicted. In case #2, if the WATLF of the cache line is "0", then the cache does no logging. In case #3, if the WATLF of the cache line is "0", then the cache performs the same operations that it performs in case #1. The directory controller at the request of the local cache can perform the copy into the counter buffer in parallel with either (1) acknowledging the remote invalidation request (in case #1) or (2) sending the dirty data to the remote cache (in case #3). In other words, the delay for the normal activities to maintain cache coherence masks the delay for copying the counter into the counter buffer.

In case #2 and #3, if the WATLF flag of the matching cache line is "1", then the cache does the following. The cache copies the extended tags and the counters of all the cache lines with non-zero counters into the counter buffer. Of course, the DRSM-L sets the SEF in the memory module to "1" only for the counter of the cache line that will be evicted. Since the logging of counters for case #2 and #3 is a total logging of all non-zero counters, the cache resets all the WATLFs to "0".

The aim of this elaborate scheme to log counters into the counter buffer is to effectively record when a line is evicted from the local cache. During

recovery, the DRSM-L loads the counter and SEF in the 2nd-level cache with the value of the counter and SEF logged in the counter buffer. When a processor hits in the 1st-level cache, it forwards a hit notice to the 2nd-level cache; it then decrements the counter of the matching cache line. Once the counter reaches zero, if the SEF is set to "1", then the DRSM-L (1) knows that an eviction occurred during normal execution (prior to recovery), (2) invalidates the cache line, and (3) re-loads it with the next line and next counter and SEF from the line buffer and counter buffer, respectively. If the SEF is "0", then the DRSM-L knows that an invalidation did not occur and simply re-loads the counter and SEF of the cache line with the next matching entry (according to the extended tag) from the counter buffer.

To reduce the number of counters that are copied into the counter buffer, we introduced the WATLF. The WATLF enables the cache to avoid stalling the processor to write all the non-zero counters into the counter buffer. If the WATLF is "0", then the counters needed to reproduce the dirty data to be delivered to a remote cache or to be written back into memory (in case #2 and #3) have already been saved in the counter buffer. So, the DRSM-L saves the counter of only the evicted cache line into the counter buffer and, for case #3, sets the SEF. On the other hand, if the WATLF is "1", then the necessary counters have not yet been saved into the counter buffer, so the DRSM-L must write all nonzero counters into the counter buffer.

D. Establishing Checkpoints

If a processor, say "P[1]", wishes to establish a checkpoint, "P[1]" first updates the checkpoint-state-buffer (CSB) of the PSB to "TENTATIVE_CHECKPOINT_IS_ACTIVE", indicating that "P[1]" is in phase 1, the tentative checkpoint. Then, "P[1]" waits until all its pending memory accesses are complete (or negatively acknowledged). Next, "P[1]" writes all its dirty blocks in the 2nd-level cache into the TCB. "P[1]" downloads its internal registers (i. e. the processor state) into the tentative-checkpoint area (not shown in figure 13) of the PSB (while preserving the previous permanent checkpoint). At the end of phase 1, "P[1]" updates the CSB of the PSB to "PERMANENT_CHECKPOINT_IS_ACTIVE", indicating that "P[1]" is now in phase 2, the permanent checkpoint.

"P[1]" writes all the dirty blocks in the 2nd-level cache back into main memory, which always holds the permanent checkpoint. "P[1]" resets all the index registers in both the local memory module and the 2nd-level cache. "P[1]" resets all the WATLFs to "0". "P[1]" then tells the PSB to invalidate the old permanent checkpoint in the PSB and to designate the processor state saved in the tentative-checkpoint area as the new permanent checkpoint. Finally, "P[1]" updates the CSB of the PSB to "CHECKPOINT_IS_NOT_ACTIVE", indicating that phase 2 (and the entire checkpoint) is finished.

E. 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(E) = amount of silicon area consumed by transistors to implement "E" entries
AA = fixed amount of allocated silicon area in which to implement the counter buffer and the line buffer
CBR = rate at which counter buffer fills in terms of the number of entries per unit time
LBR = rate at which line buffer fills in terms of the number of entries per unit time
CR = rate of establishing checkpoints

Then, ignoring other triggers for checkpoints, we have the following equations.

```
A(E[CB]) + A(E[LB]) = AA                          (equation #1)
CR = max (CBR / E[CB], LBR / E[LB])               (equation #2)
```

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

```
CR = max (CBR / E0[CB], LBR / E0[LB])             (equation #3)
   = CBR / E0[CB] = LBR / E0[LB].                 (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[CL]" must decrease to some value, say "E2[LB]. Then, we have that

```
CR = max (CBR / E2[CB], LBR / E2[LB])             (equation #5)
   = LBR / E2[LB].                                (equation #6)
```

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

```
CR = max (CBR / E1[CB], LBR / E1[LB])             (equation #7)
   = CBR / E1[CB].                                (equation #8)
```

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

```
E[CB] / E[LB] = CBR / LBR.                        (equation #9)
```

F. Summary

DRSM-L differs fundamentally from the DRSM, DRSM-C, and DRSM-H in that a processor can establish a checkpoint independently of all other processors. A processor in the DSMM with DRSM-L has this flexibility since the processor ensures that dirty data sent to other processors is never lost even if the sending processor rolls back to the previous checkpoint. Specifically, each processor logs each incoming cache line into the line buffer of the local DRSM-L module and logs the number of accesses (that the processor makes) to this cache line into the counter buffer of the DRSM-L module. If the processor encounters a fault and rolls back to the last checkpoint, the

processor uses the cache data and access history recorded in the line buffer and the counter buffer to satisfy all cache accesses during the recovery period. Once the counter buffer is exhausted, the recovery is complete, and the processor resumes normal execution. In this way, the processor re-calculates exactly the original values of the dirty data that the processor sent to other processors prior to encountering the fault. We note that DRSM-L also enables a processor to roll back to the last checkpoint independently of all other processors.

IX. Recovery

Fault-tolerance schemes generally involve 2 phases: (1) establishing periodic checkpoints or logging data 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. Although the focus of our work is the establishment of checkpoints, we comment briefly on the issue of rolling back from a fault for each of our 4 hardware-based algorithms.

A. DRSM-C

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 the spare processor. Then, RLC resets the processor, say "P3", and directs it to begin the recovery activity. "P3" negatively acknowledges all cache-coherence messages from the directory controllers until recovery is complete. "P3" invalidates all entries in its cache and tells all the directory controllers to update their directories to indicate that "P3" no longer has any cache lines. "P3" checks its PSB. If "P3" failed during the establishment of a permanent checkpoint, then "P3" completes the permanent checkpoint, telling the memory modules to transition the 3BSRs from states "[W, PC]", "[PC, W]", "[TC, PC]", and "[PC, TC]" to "[I, PC]", "[PC, I]", "[PC, I]", and "[I, PC]", respectively, for all blocks where "P3" is the active writer; "P3" tells the PSB to designate the processor state saved in the tentative-checkpoint area as the permanent checkpoint. Otherwise, if "P3" failed during the establishment of a tentative checkpoint, then "P3" discards the tentative checkpoint, telling the memory modules to transition the 3BSRs from states "[W, PC]", "[PC, W]", "[TC, PC]", and "[PC, TC]" to "[I, PC]", "[PC, I]", "[I, PC]", and "[PC, I]", respectively, for all blocks where "P3" is the active writer; "P3" tells the PSB to invalidate the processor state saved in the tentative-checkpoint area. "P3" then loads the processor state stored in the permanent-checkpoint area of the PSB and resumes execution.

B. DRSM

The recovery procedure for DRSM is similar to that for DRSM-C. The principal difference is that "P3" must query the dependency matrix of each DRSM module in order to determine all other processors that must also roll back to the last permanent checkpoint. Reversing the arrows in expression #1, #2, #3, and #4 yields the recovery dependencies. "P3" spawns a recovery tree that is similar to the checkpoint tree (i. e. processor tree) generated during the establishment of a tentative checkpoint.

DRSM has a scenario that does not arise for DRSM-C. "P3" can be establishing a checkpoint with several other dependent processors when "P3" encounters a fault. If "P3" fails during the establishment of the permanent checkpoint, "P3" completes the permanent checkpoint along with the other dependent processors. "P3" can immediately resume normal execution. If "P3" fails during the establishment of the tentative checkpoint, "P3" along with all the other processors in the checkpoint tree must discard this tentative checkpoint and roll back to their previous permanent checkpoints. The roll-back of each processor generates its own recovery tree of processors; all the recovery trees can be combined into 1 huge recovery tree where all participating processors can recover concurrently.

C. DRSM-H

The recovery procedure for DRSM-H is similar to that for DRSM. The principal differences are the following. If "P3" failed during the establishment of a permanent checkpoint, then "P3" must read the associated TCB and write all dirty blocks back into main memory. Then, "P3" must tell all memory modules to transition each 2BSR to state "PC". On the other hand, if "P3" failed during the establishment of the tentative checkpoint, then "P3" only tells all memory modules to transition each 2BSR to state "PC".

D. DRSM-L

The recovery procedure for DRSM-L differs fundamentally from the others. Suppose that the processor experiencing the fault is, again, "P3". If "P3" failed during the establishment of the permanent checkpoint, according to the PSB, then "P3" must read the associated TCB and write all dirty blocks into main memory. "P3" then completes the establishment of the permanent checkpoint that was in progress when the fault occurred. Then, "P3" queries all the memory modules to find messages which were sent to "P3" just prior to the fault; "P3" negatively acknowledges them. "P3" then resumes normal processing.

If "P3" did not fail during the establishment of the permanent checkpoint, then "P3" must perform the following procedure. "P3" queries all the memory modules to find messages which were sent to "P3" just prior to the fault; "P3" negatively acknowledges them. Then, "P3" reads the entire line buffer and the entire counter buffer and groups the entries according to the cache index of the extended tag so that the entries can be easily fetched on a miss in the 2nd-level cache. "P3" 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, "P3" invalidates its caches and loads the processor state saved in the permanent- checkpoint area of the PSB. "P3" resumes execution in recovery mode. In this mode, if a miss occurs 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 and places the line and counter into the cache. The VMM also sets the SEF in the cache to the value stored in the SEF of the SCB. Each subsequent hit in the 1st-level cache causes the corresponding counter in the 2nd-level cache to be decremented. Once a hit causes the counter to underflow, then a trap occurs to the VMM. It checks the SEF of the cache line. If the SEF is "0", then the VMM retrieves the next matching counter from the SCB to insert into the cache line. If the SEF is "1" (indicating that the cache line in normal processing was evicted from the cache), then the VMM retrieves both the next matching line from the SLB and the next matching counter. Eventually, the VMM will attempt to retrieve a counter from an empty SCB. When that event

happens, the "P3" will be in the state just prior to the occurrence of the fault, and recovery is complete. The VMM then places "P3" in the normal mode of execution where the counter increments on each hit. During the recovery mode, "P3" negatively acknowledges all messages from the rest of the DSMM. 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.

X. Implementation of Checkpoint Algorithms in ABSS

A. Communication between the DSMM and its Environment

We do not simulate interaction between the DSMM and its environment. Such communication forces the establishment of a checkpoint for any processor involved in the communication. The steps involved in establishing this checkpoint are virtually identical to the steps involved in establishing the checkpoint for the other triggers. So, understanding the impact of checkpoints initiated by the reduced set of triggers allows us to understand the impact of checkpoints initiated by all the triggers.

B. Simulation of the 3-bit State Register (3BSR)

Faithfully simulating the 3-bit state register (3BSR) associated with each pair of blocks in memory requires that ABSS perform 2 separate scans of all blocks of memory. One scan occurs during the establishment of the tentative checkpoint; ABSS must transition the state from "[W, PC]" or "[PC, W]" to "[TC, PC]" or "[PC, TC]", respectively. The second scan occurs during the establishment of the permanent checkpoint; ABSS must transition the state from "[TC, PC]" or "[PC, TC]" to "[PC, I]" or "[I, PC]", respectively. Implementing such a scan in software can significantly reduce the speed of the simulation by several orders of magnitude. Hence, we choose to not implement the scan itself. Rather, we emulate the effect (of the 3BSR) that is seen by the rest of the DSMM. To accomplish our goal, we use the checkpoint counters from the original RSM algorithm. The correspondence between the states of the 3BSR and the states derived from the combination of the global checkpoint counter (GCC), the local checkpoint counter (LCC), and the processor lock (PL) is the following.

1. [I, PC] <==> (LCC[last_writer] != GCC[last_writer])
2. [PC, I] <==> (LCC[last_writer] != GCC[last_writer])
3. [W, PC] <==> (LCC[last_writer] == GCC[last_writer]) && (PL[last_writer] == 0)
4. [PC, W] <==> (LCC[last_writer] == GCC[last_writer]) && (PL[last_writer] == 0)
5. [TC, PC] <==> (LCC[last_writer] == GCC[last_writer]) && (PL[last_writer] != 0)
6. [PC, TC] <==> (LCC[last_writer] == GCC[last_writer]) && (PL[last_writer] != 0)

The correspondence between the transitions of the states of the 3BSR and the transitions of the combination of the GCC, the LCC, and the PL are the following.

1. [I, PC] --> [W, PC] <==> LCC[last_writer] <-- GCC[last_writer]
2. [PC, I] --> [PC, W] <==> LCC[last_writer] <-- GCC[last_writer]
3. [W, PC] --> [TC, PC] <==> (PL[last_writer] <-- 1) && (LCC[last_writer] == GCC[last_writer]))
4. [PC, W] --> [PC, TC] <==> (PL[last_writer] <-- 1) && (LCC[last_writer] == GCC[last_writer]))
5. [TC, PC] --> [PC, I] <==> GCC[last_writer] <-- 1 + GCC[last_writer]
6. [PC, TC] --> [I, PC] <==> GCC[last_writer] <-- 1 + GCC[last_writer]

The DRSM (or DRSM-C) can execute a copy-on-write in parallel with satisfying a write access; hence, the DRSM can hide the time used by the copy-on-write. During a write access to a block in state "[PC, I]", the DRSM must read data from the left block (in state "PC) and return the data to the processor submitting the access. When the DRSM returns the data to the processor, the DRSM can simultaneously send a copy into the right block (in state "I"), transitioning its state to "W". The DRSM effectively uses the time for answering the write access to hide the time used by the copy-on-write. Hence, we do not explicitly modify ABSS to simulate the behavior of copy-on-write or even to simulate the existence of 2 banks of memory.

Therefore, we can collapse the 6 valid states and 6 valid state-transitions of the 3BSR into 3 valid states and 3 valid state-transitions. There is a one-to-one correspondence between these 3 valid states and the 3 valid states derived from the combination of the GCC, the LCC, and the PL for each last writer. There may be more than 3 states of the combination, but only 3 states are valid.

Based on the 3 valid states derived from this combination, we modify ABSS so that the DRSM (or DRSM-C) performs the same activities that it would perform for the corresponding 3 valid states of the 3BSR. From the point of view of the rest of the DSMM, a DRSM (or DRSM-C) using checkpoint counters and a lock array functions identically to a DRSM using the 3BSR.

C. Simulation of the 2-bit State Register (2BSR)

By similar reasoning, we can use the GCC, the LCC, and the PL to simulate the 2-bit state register (2BSR) of the DRSM-H. The correspondence between the states of the 2BSR and the states derived from the combination of the GCC, the LCC, and the PL is the following.

1. [PC] <==> (LCC[last_writer] != GCC[last_writer])
2. [W] <==> (LCC[last_writer] == GCC[last_writer]) && (PL[last_writer] == 0)
3. [TC] <==> (LCC[last_writer] == GCC[last_writer]) && (PL[last_writer] != 0)

The correspondence between the transitions of the states of the 2BSR and the transitions of the combination of the GCC, the LCC, and the PL are the following.

1. [PC] --> [W] <==> LCC[last_writer] <-- GCC[last_writer]
2. [W] --> [TC] <==> (PL[last_writer] <-- 1) && (LCC[last_writer] == GCC[last_writer]))
3. [TC] --> [PC] <==> GCC[last_writer] <-- 1 + GCC[last_writer]

XI. Results

A. Parameters and Benchmarks

We implemented each of DRSM-C, DRSM, DRSM-H, and DRSM-L in our simulated DSMM. We set the timer (that alerts a processor to establish a checkpoint) to expire after 20 million cycles (according to the processor clock) since the establishment of the last checkpoint. Twenty million cycles is 0.1 second. We set the line buffer in the DRSM-L to 4096 entries and the counter buffer to 16384 entries; the ratio of the number of entries in the counter buffer to the number of entries in the line buffer is 4.

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 over time among water molecules.

These benchmarks represent a wide variety of memory-access patterns but do virtually no communication with the environment of the DSMM. So, establishing a checkpoint that is triggered by communication between a processor and the environment outside of the DSMM does not arise in our simulations. We note that regardless of the event (e. g. expiration of a timer, overflow of the line buffer, etc.) triggering the establishment of a checkpoint, the procedure for establishing a checkpoint remains the same. Hence, we can still ascertain the performance of our hardware-based algorithms even if checkpoint establishment is triggered by a smaller set of events.

B. Checkpoints and Logs

Table 1 below shows the number of checkpoints established per processor for each of DRSM-C, DRSM, DRSM-H, and DRSM-L. DRSM has the fewest number of checkpoints because this DRSM has only 1 trigger (i. e. expiration of a timer) for establishing a checkpoint. DRSM-C suffers the most number of checkpoints since a processor must establish a checkpoint whenever a dirty block written by that processor is accessed by another processor; the frequency of this sort of communication is rather high in, for example, "radix". DRSM-H has more checkpoints than DRSM because a group of dependent processors among which a dirty cache line is "floating" must establish a checkpoint when the last copy of that dirty cache line is evicted from the group. DRSM-L has almost as few checkpoints as DRSM has because both the line buffer and the counter buffer are large enough to reduce the occurrence of overflows to a small number. A processor must establish a checkpoint when either of these buffers overflows.

	DRSM-C	DRSM	DRSM-H	DRSM-L
cholesky	718.8	11.0	54.4	12.6
FFT	16.4	2.0	6.0	2.1
LU	141.0	9.4	11.2	8.9
ocean	2612.2	24.0	73.0	42.9
radix	390.1	1.0	16.0	4.1
water	195.5	8.0	8.0	7.6

Table 1. Checkpoints per Processor

Table 2 below shows several statistics. Each number without parentheses is the total number of dirty cache lines that are written back per processor into main memory for all the checkpoints that occurred during the run of the benchmark. For DRSM-C, DRSM, and DRSM-H, each number within parentheses indicates the total number of dirty memory blocks (where a memory block is identical in size to a 2nd-level-cache line) that are saved in the permanent checkpoint. For DRSM-C and DRSM, the number within parentheses must be larger than the number without parentheses. Dirty data that will be saved in the permanent checkpoint need not necessarily all reside in the caches. Some of that dirty data could be sitting in 1 of the 2 banks of memory. For DRSM-H, the number within parentheses is generally smaller than the number without parentheses. This behavior is due to the fact that we increased the number of states for a cache line to include DIRTY_SHARED. When a processor establishes a checkpoint, the processor will write all dirty cache lines, lines with a state of either DIRTY_SHARED or EXCLUSIVE, back into main memory. There can be multiple copies of the same line in a state of DIRTY_SHARED. Since there is no convenient way in which to synchronize the caches to write only 1 copy of a DIRTY_SHARED line back into memory, the DSMM writes all of the copies back into memory, resulting in some unnecessary write-backs.

Finally, for DRSM-L, 2 numbers reside within parentheses. The number before the semicolon is the number of cache lines logged per processor into the line buffer. The number after the semicolon is the number of cache counters logged per processor into the counter buffer. Since the ratio of the number of cache counters to the number of cache lines ranges from approximately 2.5 to approximately 5.0 across most of the benchmarks, the relative sizes of our counter buffer and our line buffer is close to optimal as the ratio of their entries is 4.

	DRSM-C	DRSM	DRSM-H	DRSM-L
cholesky	58201.5	22731.1	46831.0	24738.2
	(58210.5)	(27633.5)	(42821.1)	(35622.4; 120469.5)
LU	27198.6	9807.0	15208.6	9991.4
	(27202.1)	(11482.8)	(12007.2)	(11007.9; 34064.8)
FFT	8496.5	2193.2	11094.8	2262.2
	(8496.5)	(2318.1)	(8466.4)	(8774.9; 20562.6)
ocean	275522.6	98751.4	143959.5	128062.5
	(275533.2)	(115946.6)	(139939.8)	(124021.1; 513912.6)
radix	57779.0	1034.5	14874.0	4954.5
	(57779.0)	(1035.9)	(14327.8)	(8326.1; 64198.2)
water	5326.8	1463.4	4233.0	1442.6
	(5326.8)	(2019.1)	(2025.4)	(4776.0; 23232.6)

Table 2. Checkpoint Data and Logging Data per Processor (refer to text)

C. General Performance across the Benchmarks

To study the performance of our 6 benchmarks, we analyze the number of cycles consumed by 5 major components (of execution): non-idle time of the processor, the instruction stall, the lock stall, the data stall, and the barrier stall.

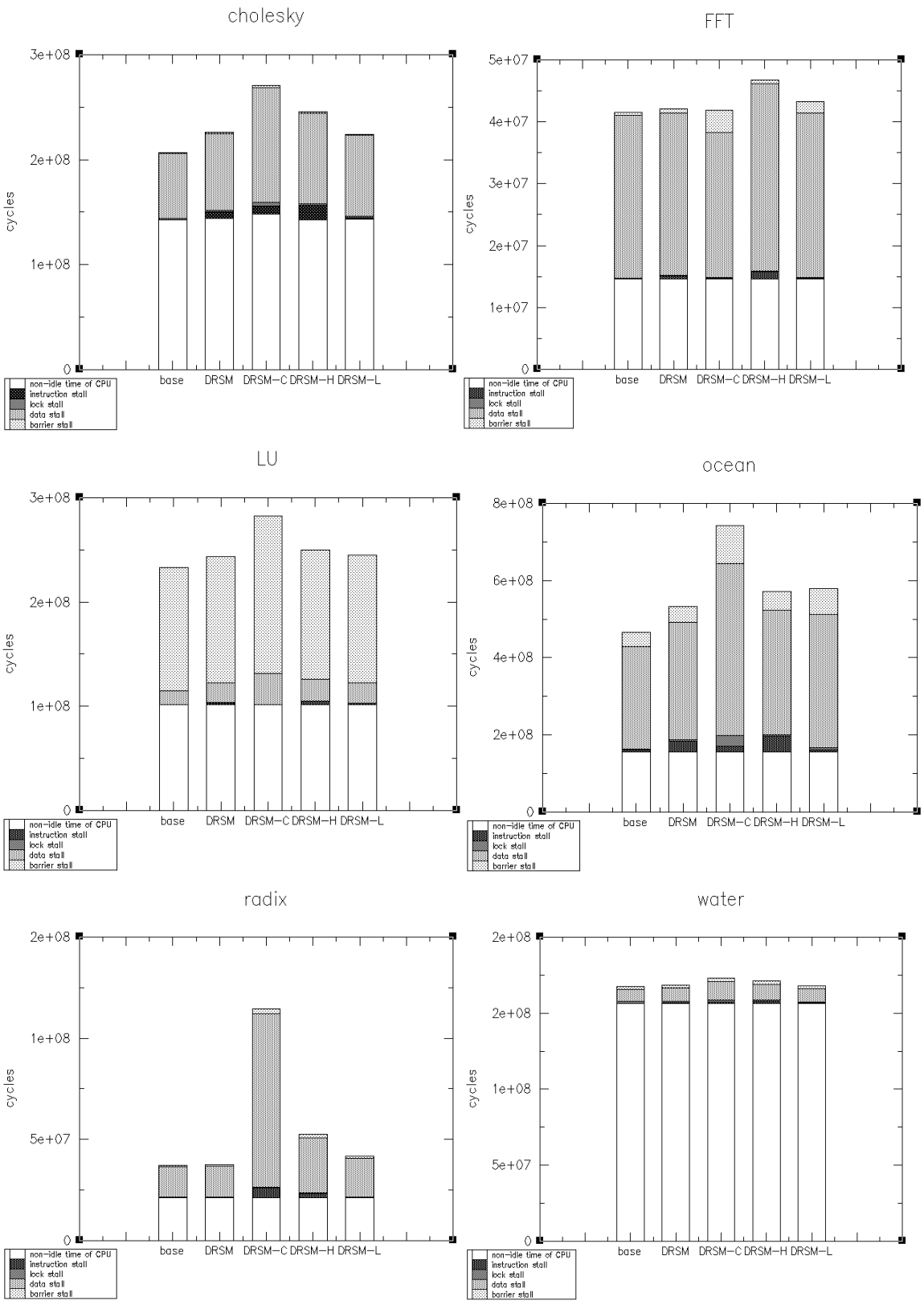

Figure 14. Cycles of Execution for the Benchmarks

The data stall includes the amount of time used for both upgrades and genuine misses. In our measurements, we obtain data for each of the 8 processors and present the average of the data for all processors except processor #0. Initialization effects skew the data for processor #0, so we exclude it.

Figure 14 summaries the performance of the base DSMM, DSMM with DRSM-C, DSMM with DRSM, DSMM with DRSM-H, and DSMM with DRSM-L for each of the benchmarks. We see 2 clear trends. Across the benchmarks, DRSM generally has the best performance among the hardware-based algorithms, and DRSM-C generally has the worst performance. Each processor in a DSMM with DRSM establishes a checkpoint under only 1 condition: expiration of the timer. A processor in the other DSMMs establishes a checkpoint under more conditions. In particular, a processor in a DSMM with DRSM-C establishes a checkpoint under the worst condition: dirty data written by that processor is accessed by another processor. In applications like radix with much dirty sharing, DRSM-C has an extremely negative impact on performance.

The notable exception is FFT running on a DSMM with DRSM-C. FFT, like radix, has much dirty sharing but performs well on DRSM-C. The execution of FFT consists of a repeated sequence of barrier, computation, barrier, and communication. During communication, each processor reads dirty data from all other processors into its own section of memory. In the base DSMM, this read of dirty data causes a delay in the form of 1 message (from the directory controller) to the processor owning the dirty data and 1 acknowledgment message from that same processor, replying with the dirty data. In the DSMM with DRSM-C, the inter-processor communication at each barrier causes all the processors to establish a checkpoint, flushing all dirty data in the caches back into main memory. By the time that the execution of the FFT reaches the communication phase (in the repeated sequence), each processor reads only data that is already clean in other processors, all dirty data already having been flushed at the previous checkpoint. Hence, the DSMM with DRSM-C avoids the cost of the acknowledgment message that the base DSMM suffers for the dirty read. (We discuss this phenomenon in detail in a subsequent section.) The elimination of this cost offsets the increase in execution time caused by the interference of checkpoint establishment on the execution of the FFT. An example of interference is flushing dirty local data (i. e. data which is not shared among processors) in the cache back into main memory. The net effect is that the FFT running on DRSM-C performs as well as the FFT running on the base system.

The performance decline for DRSM-H is due to 4 reasons. First, the dependency matrix records a doubly linked dependency for the read-after-write case (expression #3), so a processor is more likely to establish a checkpoint. Second, establishing a checkpoint can be triggered by an additional event: the last copy of a dirty cache line is evicted from the cache. Third, when a checkpoint is established, multiple processors may write multiple copies of the same dirty cache line back into memory. Fourth, after cold-cache misses are satisfied, some read misses incur the additional cost of finding a DIRTY_SHARED copy in a remote cache since DRSM-H tries to avoid write-backs into memory in order to minimize the number of checkpoints. The last reason is likely the principal cause of the performance decline as the number of checkpoints established for DRSM-H is not significantly larger than the number of checkpoints established for DRSM.

The performance decline for DRSM-L is due to 1 major and 1 minor reason. The minor reason is that the line buffer or the counter buffer occasionally

overflows. When either of them overflow, the processor must establish a checkpoint in order to empty the buffers. In the simulated system, the buffers are sufficiently large that overflows occur only occasionally. The major reason that DRSM-L performs worse than DRSM is that whenever a local processor sends a copy of a dirty cache line with WATLF being "1" to a remote processor (or evicts a dirty cache line with WATLF being "1"), the local processor must first perform the costly operation of copying all the nonzero counters into the counter buffer. The delay for this copy is seen by both the local processor and the remote processor. Worse, if other cache accesses (e. g. invalidations) from remote processors arrive during this copying of counters into the counter buffer, those cache accesses must wait in the queue until the copying is finished. The local processor also incurs additional delay if the processor evicts a dirty cache line with its WATLF being "1" on a conflict miss.

Finally, the base DSMM, DSMM with DRSM-C, DSMM with DRSM, DSMM with DRSM-H, and DSMM with DRSM-L all perform almost equally well in running water. There are 2 reasons for this behavior. Water has little communication among the processors (helping DRSM-C and DRSM-L), and the working set of data fits within the 2nd-level cache (helping DRSM-H).

D. Performance of a Specific Benchmark

We now focus on behavior that is peculiar to each benchmark.

1. Fast Fourier Transform (FFT)

Compared to the base DSMM, the system with DRSM performs well because it minimally perturbs the normal operation of the DSMM. Interference occurs only during the periodic establishment of checkpoints, determined solely by the expiration of a timer. DRSM-H performs noticeably worse probably because a substantial number of reads misses occur on memory blocks that are held in a cache state of DIRTY_SHARED. If a block is DIRTY_SHARED, the directory controller must retrieve a copy of the block from a processor holding a copy of it. Retrieving a copy from a processor requires more time than merely retrieving a copy from the local memory. (The number of additional checkpoints caused by evicting the last copy of a dirty block "floating" among a group of caches is not large enough to account for the decreased performance of DRSM-H.)

The surprising result is the performance of the DSMM with DRSM-C. Compared to the result for the base DSMM, the data stall decreases significantly, and the barrier stall increases by almost the amount of the decrease. FFT running on the DSMM with DRSM-C runs almost as fast as the FFT running on the base DSMM.

We find the explanation for these results in how the algorithm for FFT works [11]. It mainly runs in a repeated sequence of barrier, computation, barrier, and communication. Figure 15 presents a simplified view of the sequence. Processor "P1" reads data from region "RA", performs computation on the data, and writes the results back into "RA". After barrier "B2", processor "P1" reads data from other regions into which other processors have written the results of computation. "P1" writes this data into region "RB". Meanwhile, those other processors read all the data from "RA". After barrier "B3", "P1" performs computation (similar to that after "B1") on the data in "RB". This latter computation does not appear explicitly in figure 15. After yet another barrier (not shown in the figure), "P1" reads data from other memory regions updated by other processors and writes that data into region "RA". The cycle

repeats several times. The transpose in the FFT algorithm corresponds to the blocked reading phases like that between "B2" and "B3"; the blocked reading phases is the principal communication in the sequence of barrier, computation, barrier, and communication.

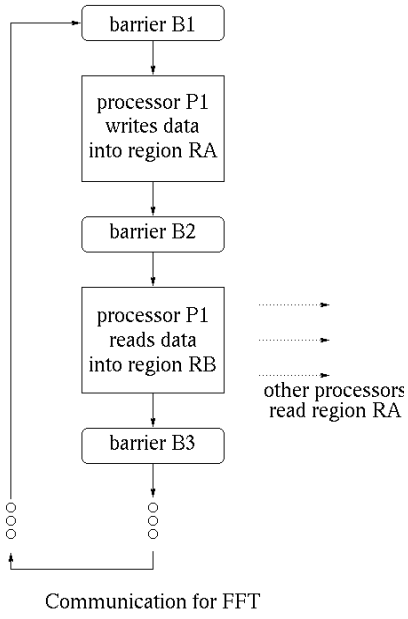

Figure 15.

The cost of the communication in figure 15 actually decreases if the DSMM uses DRSM-C. Figure 16 illustrates the messages that arise for both the base DSMM and the DSMM with DRSM-C in the context of the communication in figure 15. In the base DSMM, each read by "P2" to a dirty block generates a read request "Q1" to memory "M3". "M3" forwards the request via "Q2" to the cache of "P1". "P1" replies with "Q3". "M3" updates its copy of the dirty block, changes its state in the memory directory to "shared" (basically, the clean state), and fowards the data via "Q4" to "P2".

In the DSMM with DRSM-C, the interaction at the barrier (e. g. "B2" in figure 15) just prior to the communication causes "P1" to establish a checkpoint that transitions all blocks in "RA" into the permanent checkpoint state. The checkpoint causes all dirty blocks to be copied back into memory. The diagram in the bottom half of figure 16 uses the dashed line of "Q3" to represent this copying back of dirty data. All subsequent reads by "P2" to region "RA" (in figure 15) finds only clean blocks, and "M3" can respond to these subsequent reads. These reads cause only "Q1" and "Q4" to be generated, indicated by figure 16. Adding in the cost of "Q3", we find that DRSM-C actually reduces the cost of the communication by the amount represented by the omitted "Q2".

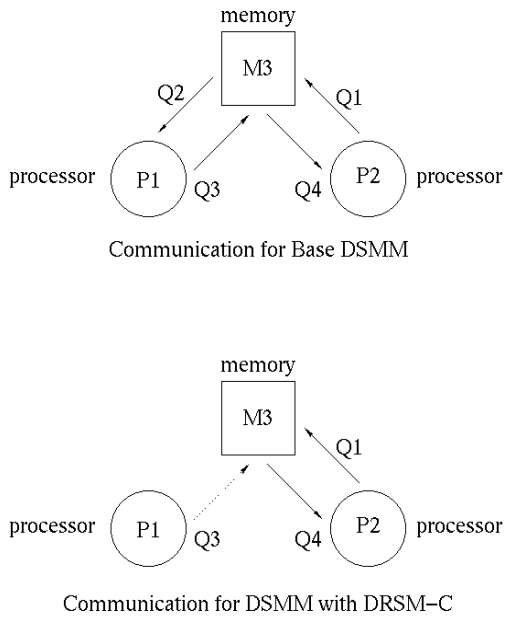

Figure 16. Messages for Communication

Hence, we see the significant reduction in data stall for DRSM-C in figure 14. A significant portion of the communication indicated in figure 15 incurs the cost of only "Q1" and "Q4". The cost of "Q3" contributes to a significant increase in the barrier stall. Eliminating the cost of "Q2" offsets the increase in data stall due to interference in the operation of the cache (since its blocks are cleaned at each checkpoint) and in other stalls due to establishing the checkpoint itself.

2. Radix

Compared to the base DSMM, the system with DRSM again performs well. DRSM-H performs noticeably worse probably because many read

misses occur on memory blocks that are in a cache state of DIRTY_SHARED. Also, DRSM-L performs noticeably worse than DRSM as radix has 1 of the 2 highest rates of filling the counter buffer and line buffer. We obtain the fill-rate by dividing the number of entries written into the counter buffer by the duration of benchmark. High fill-rates cause interference with the normal operation of the DSMM.

On the other hand, the DSMM with DRSM-C performs much worse than the other 2 systems. This poor performance is due to the nature of the algorithm for radix. Each processor runs in an iterative loop. During each iteration, each processor updates all entries in the 3 arrays accessed by all other processors. Barriers force this update to occur at approximately the same time for all processors. Different processors may update different entries that actually exist in the same cache line, resulting in false sharing, and dirty cache lines bounce back and forth among processors. Performing these writes to already dirty cache lines causes almost all the processors to establish checkpoints simultaneously and frequently (during the update of the arrays). When almost all processors establish concurrent checkpoints, they can interfere with each other by overloading the memory modules and increase the amount of time required to establish the checkpoints. The net effect is to considerably increase the duration of the data stall.

3. LU

Compared to the base DSMM, DSMM with DRSM, DSMM with DRSM-H, and DSMM with DRSM-L perform approximately equally well. The DSMM with DRSM-C performs somewhat worse than the other systems.

We look more closely at the results for DRSM-C. In each iteration of the major loop of execution, the bulk of the communication among processors occurs after the perimeter blocks are updated with the already updated diagonal block [11]. This communication consists of only each processor reading the value of the pertinent updated perimeter blocks; each processor uses them to compute the new values of its allotted group of interior blocks. The computation of the interior blocks comprises the bulk of the execution in LU. Two barriers bracket the start and the end of this computation. The interaction at the barrier signaling the start of this computation causes all processors that wrote into the perimeter blocks to establish checkpoints and hence to convert all dirty data in the perimeter blocks to clean data that is part of the permanent checkpoint. Subsequent reads to the perimeter blocks by the processors computing values for the interior blocks do not cause additional checkpoints to be established. Hence, we have an effect that is similar to that depicted in figure 16. A processor computing the new values for the interior blocks reads the value of the pertinent perimeter blocks and finds that they are already in the "clean" state, avoiding the cost of message "Q2". Unfortunately, the savings represented by "Q2" is offset by the cache interference that establishing checkpoints causes. A processor that updates the perimeter blocks also updates some interior blocks. The caches of each processor are large enough to retain the interior blocks between iterations. In the base DSMM without checkpoints, each processor finds the interior blocks to already be dirty in its own cache when that processor attempts to update the interior blocks. In the DSMM with DRSM-C, each processor that updated a perimeter block finds the interior blocks to be clean in its own cache when that processor attempts to update the interior blocks. Hence, that processor must incur the heavy cost of sending an upgrade request (MEMSYS_UPGRADE) to the memory directory to update the state of the blocks to be dirty. Thus, the DSMM with DRSM-C performs worse than the other DSMM systems.

4. Cholesky

Cholesky uses a sub-matrix algorithm to factor a sparse matrix and is similar to LU in structure and partitioning. The principal difference is that the algorithm is not globally synchronized between steps. When a processor finishes processing a block, the processor sends a notification to all other processors (in the row or column of the block) which need to read this block. Each processor spins on a lock-protected task queue and waits for blocks that the processor needs.

We observe the following about figure 14. All the systems exhibit almost no barrier stall as cholesky has no global synchronization between steps. In addition, DRSM and DRSM-L perform equally well. DRSM-H performs noticeably worse probably because many read misses occur on memory blocks that are in a cache state of DIRTY_SHARED. DRSM-C has the worst performance. The spinning at the task queue causes processors to establish a large number of checkpoints due to communication between pairs of processors.

5. Ocean

Ocean is an algorithm that simulates that movement of water in the ocean. The algorithm partitions the ocean into a grid of squares and assigns each square to a processor. It performs computation on its own square. The communication between processors occurs mainly at the boundaries of the squares.

Compared to the base DSMM, the DSMM with DRSM performs well. DRSM-H performs noticeably worse probably because many read misses occur on memory blocks that are in a cache state of DIRTY_SHARED. Also, DRSM-L performs noticeably worse than DRSM as ocean has 1 of the 2 highest rates of filling the counter buffer and line buffer. High fill-rates cause interference with the normal operation of the DSMM. DRSM-C performs worst; several barriers that enforce synchronization between horizontal phases of each time step cause many checkpoints to be established [7].

6. Water

Water is an algorithm that evaluates the forces and potentials among a system of water molecules over a period of time [7]. The molecules exist in a cubical space. The algorithm partitions it into a number of cubes and assigns each cube to a processor. Each processor performs computations on only its own cube and reads the values of molecules that exist within a cutoff radius inside neighboring cubes. Significant communication occurs when a molecule moves from one cube into another cube.

The performance of all 5 DSMMs is equally good. DRSM-H performs as well as DRSM. Since the cache-miss rate is extremely low [12], dirty blocks in DRSM-H are unlikely to be evicted back into main memory, so DRSM and DRSM-H have the same rate of establishing checkpoints. DRSM-L performs as well as DRSM. Since the amount of communication between processors is extremely low [12] and since the eviction rate is extremely low (which is implied by the low cache-miss rate), the rate at which the line buffer and cache buffer fills is low. In fact, water has the lowest such rate, thus minimizing the impact of logging data on the normal operation of the DSMM.

DRSM-C performs well because the amount of communication between processors is extremely low, so instruction-execution time (i. e. "non-idle time of CPU" in

figure 14) dominates the total execution time of the application. Eliminating the instruction-execution time, we see that DRSM-C negatively impacts the remaining execution time to the same degree that DRSM-C negatively impacts the remaining execution time in, for example, LU. The dominance of the instruction-execution time in water masks the inherently poor performance of DRSM-C.

XII. Conclusion

Based on the results, we discover that DRSM has the best performance but has 2 principal negative features. DRSM is expensive in terms of requiring 2 banks of fault-tolerant memory and has complex features: arbiter, artificially dependent processors, dependency matrix, 3BSR, etc. DRSM-L has the second best overall performance and is relatively cheap, requiring only 1 bank of fault-tolerant memory. Further, the algorithm that DRSM-L uses to establish a checkpoint and to log data (in the line buffer and the counter buffer) is less complicated than the algorithm used by DRSM. DRSM-L also has the ideal recovery property: if a processor rolls back to the last checkpoint due to a fault, other processors need not roll back to their last checkpoints. The principal negative feature of the DRSM-L is that a recovery can be somewhat slow particularly if the line buffer and the counter buffer are large (hence requiring much time for the VMM to sort their entries). We note that in general faults should occur infrequently in well-built (i. e. reliable) systems, so slow recovery after a rare occurrence of a fault should not pose a problem. We rule out DRSM-C as a viable means of establishing checkpoints as DRSM-C can perform poorly, depending on the application. DRSM-H is probably a poorer method of establishing checkpoints than DRSM-L. Although DRSM-H performs only slightly worse than DRSM-L, DRSM-H uses a very complicated algorithm to establish checkpoints. In addition to the complexity of DRSM, DRSM-H must deal with a TCB, an additional cache-line state, etc.

Therefore, with regards to cost and performance, we conclude that DRSM-L is the best hardware-based algorithm for establishing checkpoints.

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 Symposium 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. 1101-1115, 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. P. Singh, W. D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory", technical report: csl-tr-92-526, Stanford University, pp. 1-42, June 1992.
8. D. Sunada, D. Glasco, M. Flynn, "ABSS v2.0: a SPARC Simulator", technical report: csl-tr-98-755, Stanford University, pp. 1-27, April 1998.
9. D. Sunada, D. Glasco, M. Flynn, "Fault Tolerance: Methods Of Rollback Recovery", technical report: csl-tr-97-718, Stanford University, pp. 1-51, March 1997.
10. G. Suri, B. Janssens, et. al., "Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory", Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems, pp. 279-288, 1995.
11. S. C. Woo, J. P. Singh, J. L. Hennessy, "The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors", Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS - VI), pp. 219 - 229, October 1994.
12. S. C. Woo, M. Ohara, E. Torrie, J. Singh, A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", Proceedings of the 22nd Annual International Symposium 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. 231-240, April 1990.